Deep Learning with CUDA

Laboratory 04

Andrzej Świętek

Faculty of Physics and Applied Informatics

AGH

Contents

Introduction

In this lab session, our primary goal was to investigate the effects of different Batch Norms and Dropouts layers in AlexNet as well as VGG models' training and quality of the result.

All of following tests and experiments were conducted on Fashion MNIST or CIFAR-10 datasets. All the models were built with Tensorflow (Keras).

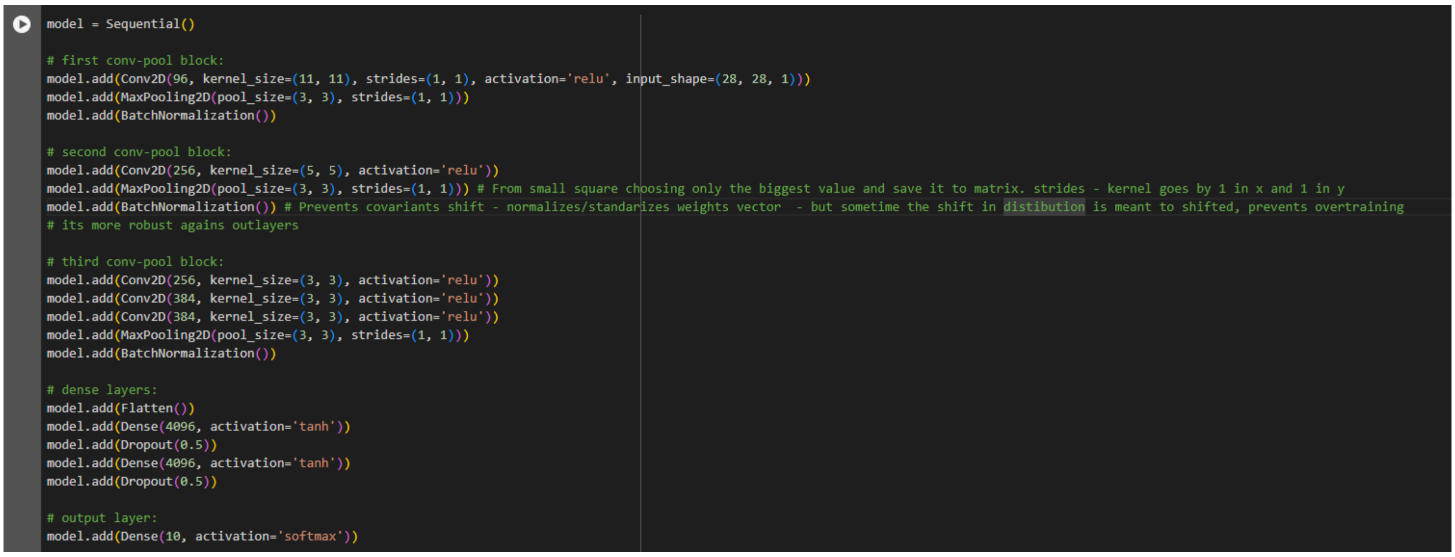
Fashion MNIST

The Fashion MNIST dataset consists of 60,000 training samples and 10,000 test samples, each with 784 features representing pixel values (28x28 images) and one label indicating the class of the fashion item. In order to identify the coordinates of a pixel in an image, consider expressing the pixel index, denoted as x as $x := i*28 + j$ where $i, j \in [0, 27]$. This representation indicates that the pixel is positioned at the intersection of the i-th row and the j-th column within a matrix of size 28 x 28. For instance, when referring to pixel 31, it signifies the pixel's location in the fourth column from the left and the second row from the top, maintaining a clear access. The dataset contains 10 classes, each associated with a specific fashion category.

Label	Category
0	T-shirt/Top
1	Trousers
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneakers
8	Bag
9	Ankle boot

Random Samples from Each Class
Class 2
Class 9
Class 6
Class 0
Class 3
Class 4
Class 5
Class 8
Class 7
Class 1

Default AlexNet Model vs Fashion MNIST



```
model = Sequential()

# first conv-pool block:
model.add(Conv2D(96, kernel_size=(11, 11), strides=(1, 1), activation='relu', input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(3, 3), strides=(1, 1)))
model.add(BatchNormalization())

# second conv-pool block:
model.add(Conv2D(256, kernel_size=(5, 5), activation='relu'))
model.add(MaxPooling2D(pool_size=(3, 3), strides=(1, 1))) # From small square choosing only the biggest value and save it to matrix. strides - kernel goes by 1 in x and 1 in y
model.add(BatchNormalization()) # Prevents covariants shift - normalizes/standarizes weights vector  - but sometime the shift in distibution is meant to shifted, prevents overtraining
# its more robust agains outlayers

# third conv-pool block:
model.add(Conv2D(256, kernel_size=(3, 3), activation='relu'))
model.add(Conv2D(384, kernel_size=(3, 3), activation='relu'))
model.add(Conv2D(384, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(3, 3), strides=(1, 1)))
model.add(BatchNormalization())

# dense layers:
model.add(Flatten())
model.add(Dense(4096, activation='tanh'))
model.add(Dropout(0.5))
model.add(Dense(4096, activation='tanh'))
model.add(Dropout(0.5))

# output layer:
model.add(Dense(10, activation='softmax'))
```

AlexNet is a deep convolutional neural network architecture comprising eight layers: five convolutional layers followed by three fully connected layers.

Convolutional Layers:

- The first block includes a 96-filter 11x11 convolutional layer followed by max-pooling and batch normalization.
- The second block features a 256-filter 5x5 convolutional layer, followed by max-pooling and batch normalization.
- The third block consists of three 3x3 convolutional layers with 256, 384, and 384 filters respectively, followed by max-pooling and batch normalization.

Dense Layers:

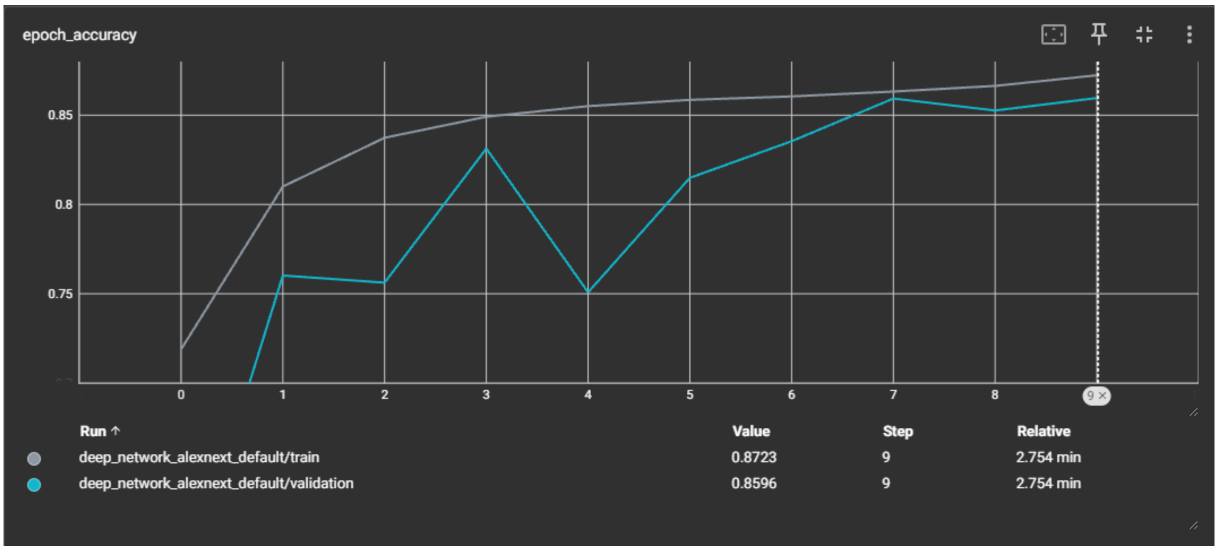
- Two dense layers with 4096 neurons each and a tanh activation function are followed by dropout layers with a rate of 0.5 to prevent overfitting.

Output Layer:

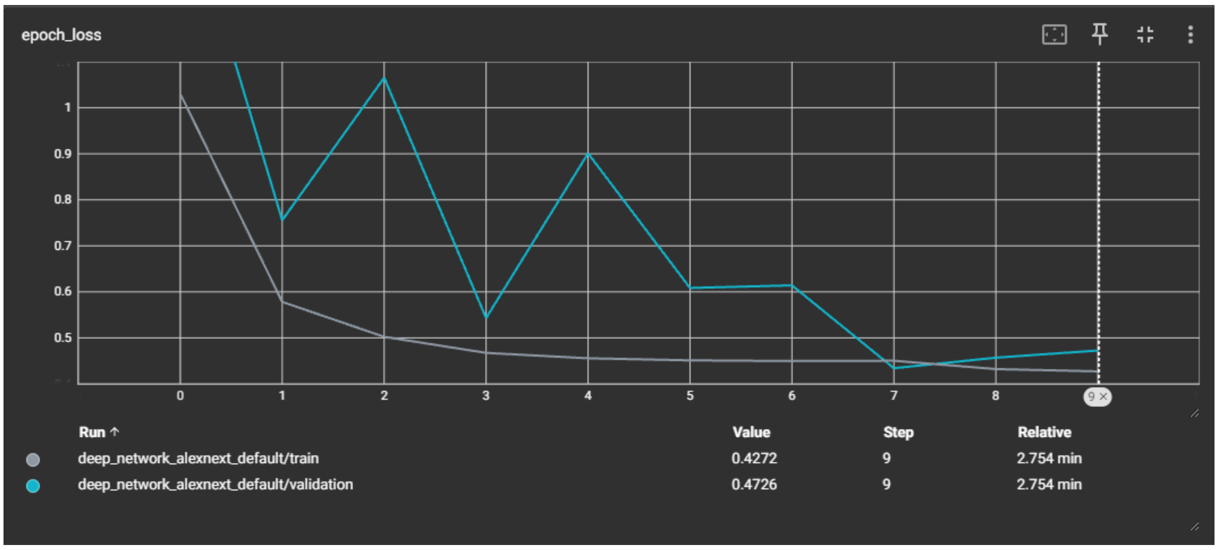
- The output layer comprises ten neurons with softmax activation for class probability estimation.

Key Divisions:

- Convolutional layers extract hierarchical features.
- Dense layers capture high-level representations.
- Batch normalization and dropout layers enhance training stability and prevent overfitting.



	Run ↑	Value	Step	Relative
●	deep_network_alexnext_default/train	0.8723	9	2.754 min
●	deep_network_alexnext_default/validation	0.8596	9	2.754 min

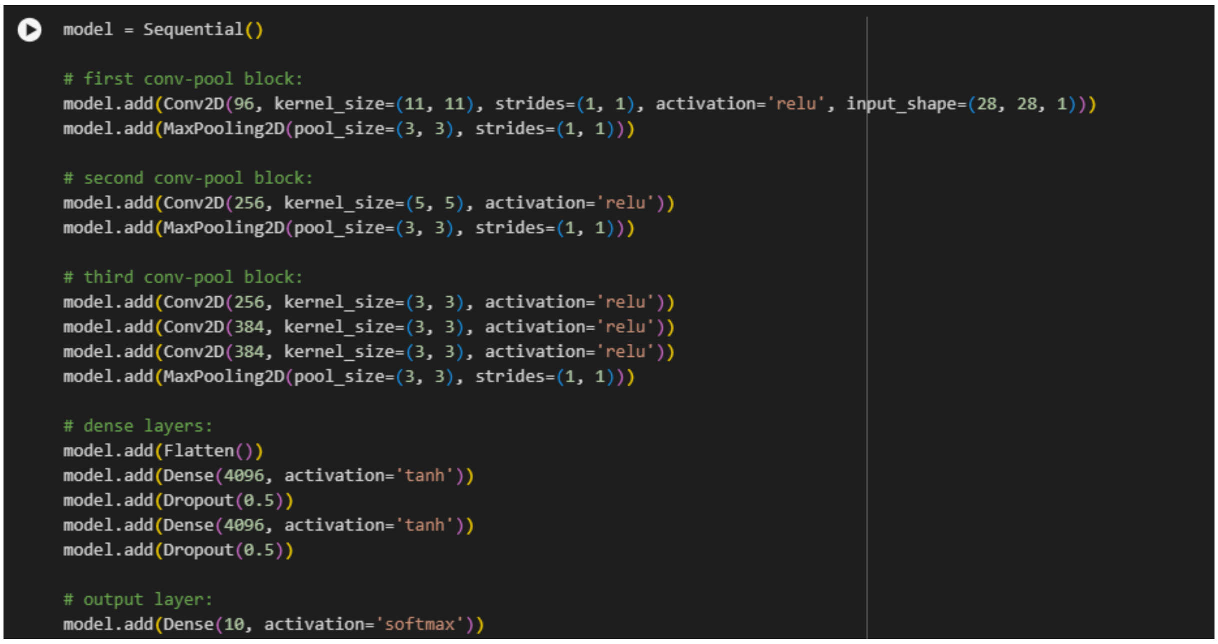


	Run ↑	Value	Step	Relative
●	deep_network_alexnext_default/train	0.4272	9	2.754 min
●	deep_network_alexnext_default/validation	0.4726	9	2.754 min

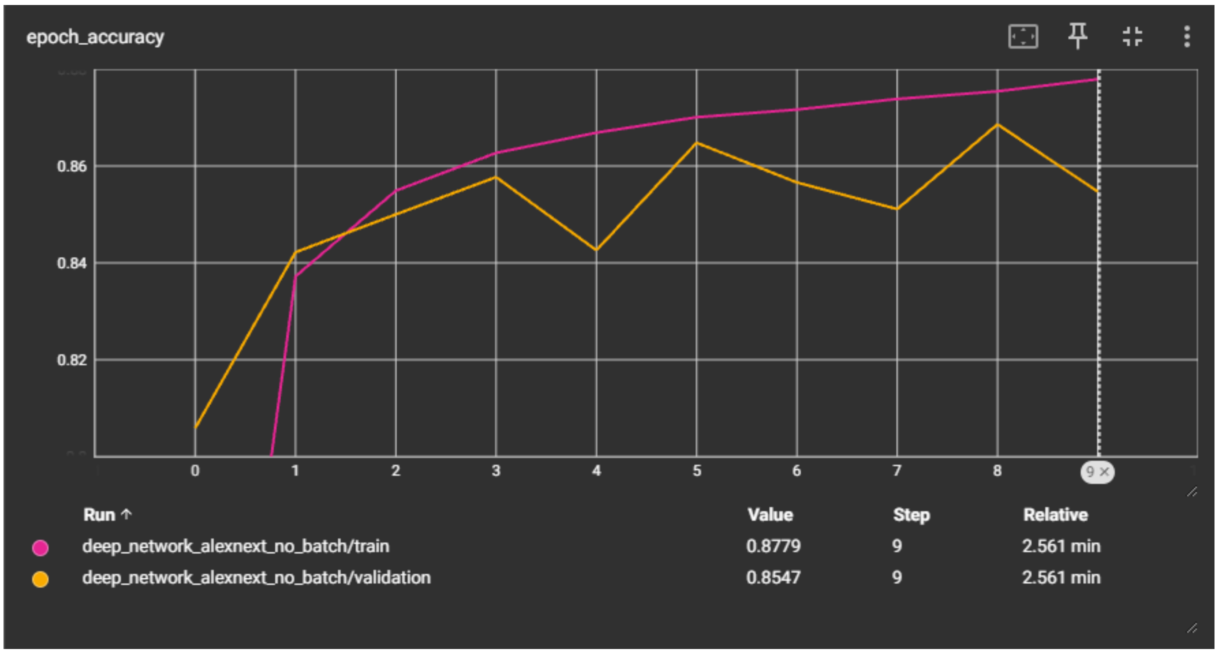
Final Validation Accuracy Score: 0.8596

No Batch Normalization after each Conv Block

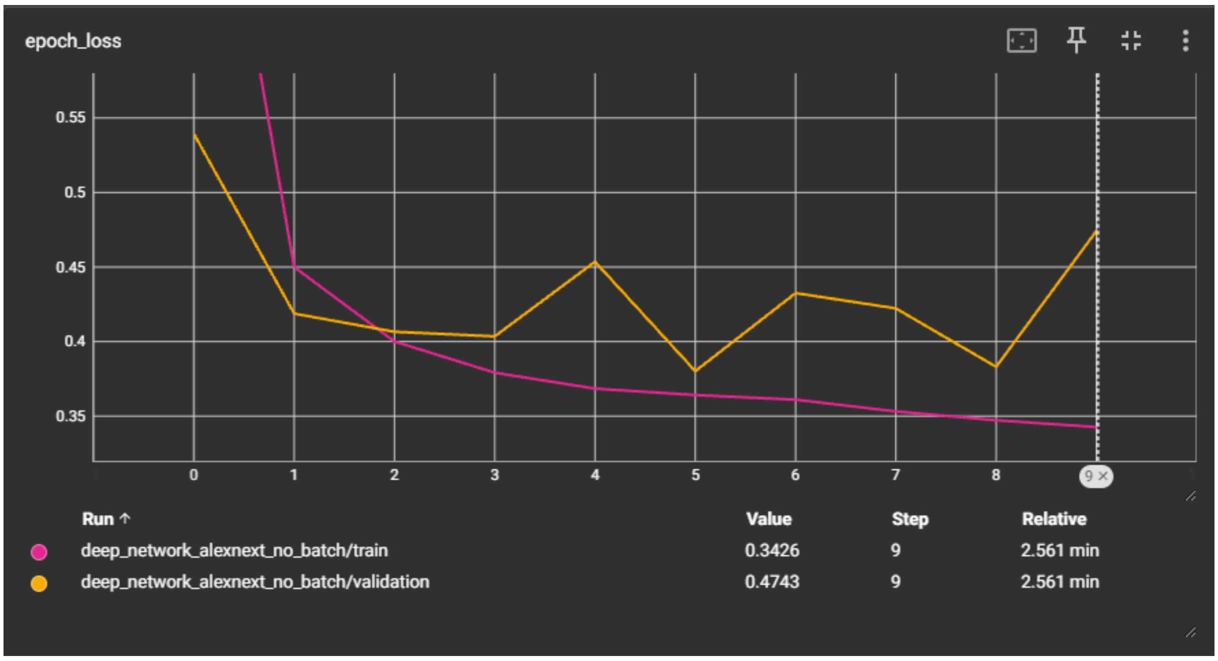
This attempt should clearly show what batch Normalization brings to the table an if its any good at all.



```
model = Sequential()

# first conv-pool block:
model.add(Conv2D(96, kernel_size=(11, 11), strides=(1, 1), activation='relu', input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(3, 3), strides=(1, 1)))

# second conv-pool block:
model.add(Conv2D(256, kernel_size=(5, 5), activation='relu'))
model.add(MaxPooling2D(pool_size=(3, 3), strides=(1, 1)))

# third conv-pool block:
model.add(Conv2D(256, kernel_size=(3, 3), activation='relu'))
model.add(Conv2D(384, kernel_size=(3, 3), activation='relu'))
model.add(Conv2D(384, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(3, 3), strides=(1, 1)))

# dense layers:
model.add(Flatten())
model.add(Dense(4096, activation='tanh'))
model.add(Dropout(0.5))
model.add(Dense(4096, activation='tanh'))
model.add(Dropout(0.5))

# output layer:
model.add(Dense(10, activation='softmax'))
```



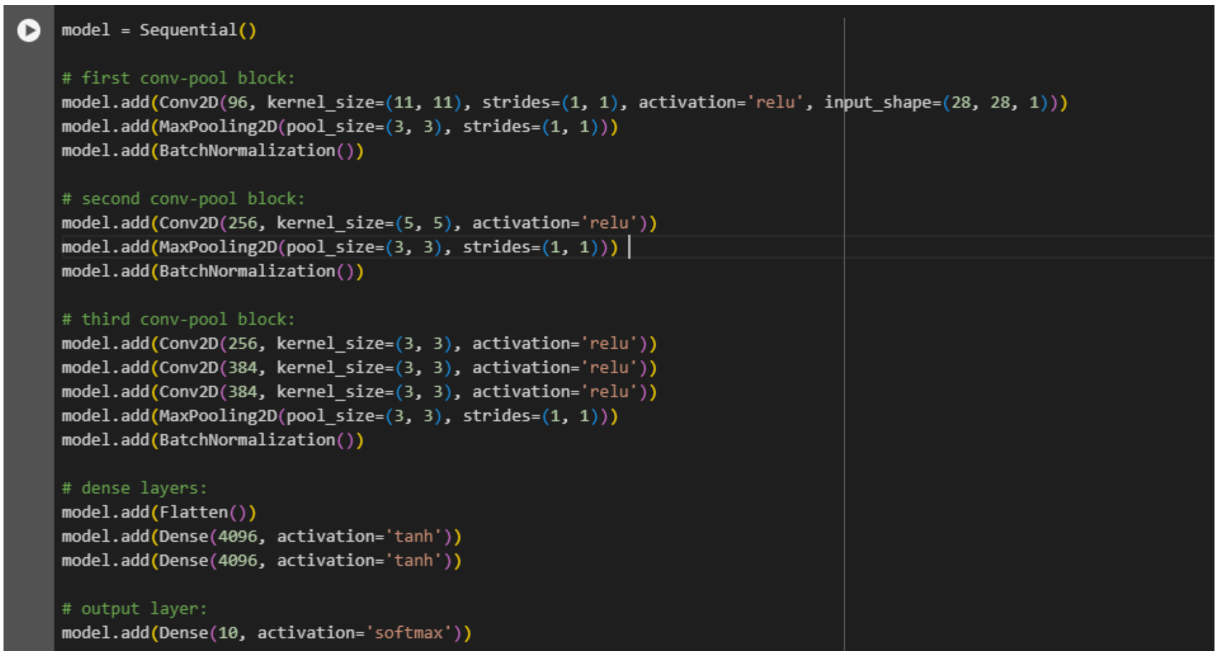
	Run ↑	Value	Step	Relative
●	deep_network_alexnext_no_batch/train	0.8779	9	2.561 min
●	deep_network_alexnext_no_batch/validation	0.8547	9	2.561 min



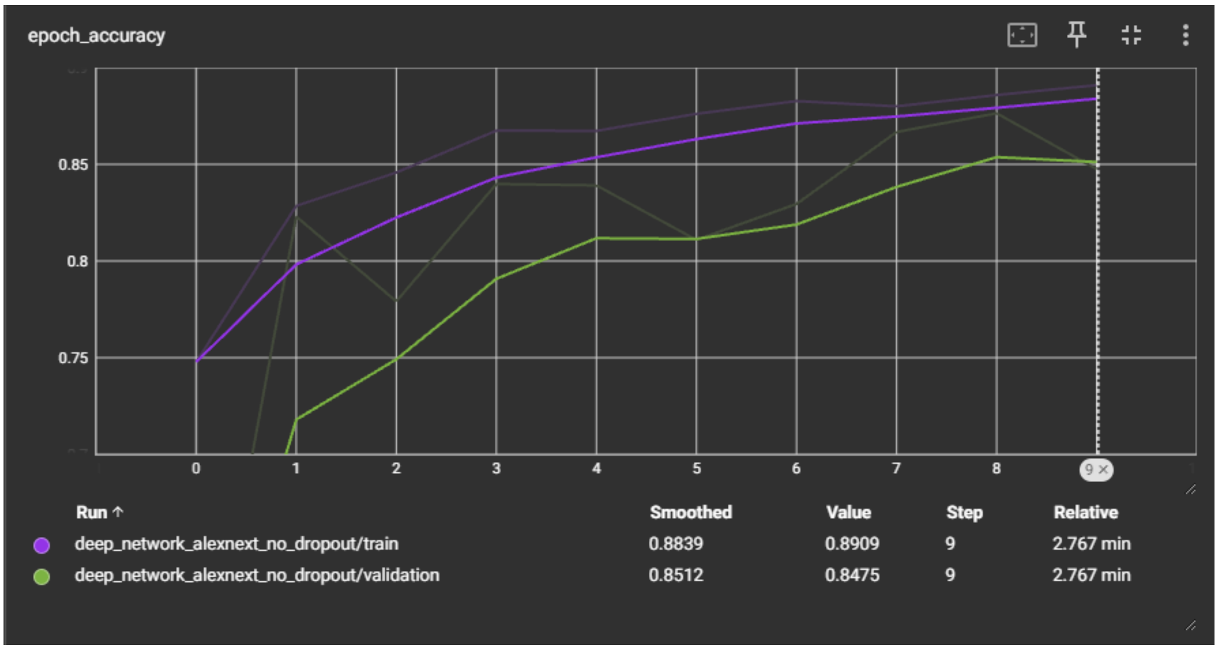
	Run ↑	Value	Step	Relative
●	deep_network_alexnext_no_batch/train	0.3426	9	2.561 min
●	deep_network_alexnext_no_batch/validation	0.4743	9	2.561 min

Final Validation Accuracy Score: 0.8547

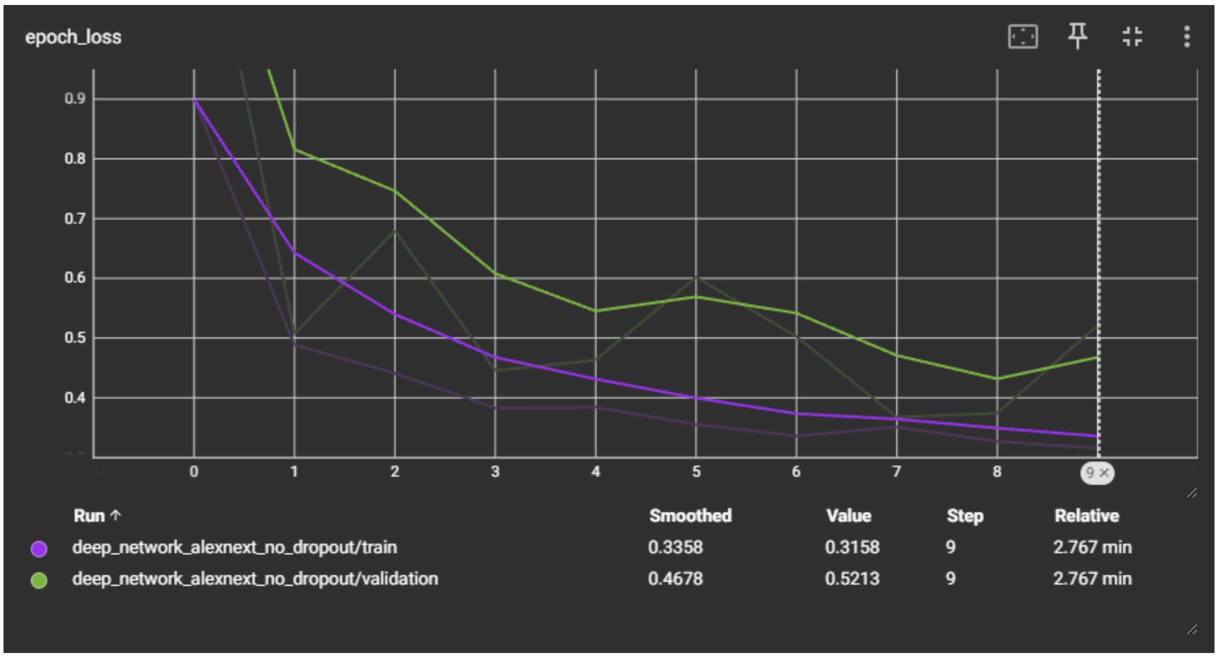
No Dropouts in Flattened part of model



```
model = Sequential()

# first conv-pool block:
model.add(Conv2D(96, kernel_size=(11, 11), strides=(1, 1), activation='relu', input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(3, 3), strides=(1, 1)))
model.add(BatchNormalization())

# second conv-pool block:
model.add(Conv2D(256, kernel_size=(5, 5), activation='relu'))
model.add(MaxPooling2D(pool_size=(3, 3), strides=(1, 1)))
model.add(BatchNormalization())

# third conv-pool block:
model.add(Conv2D(256, kernel_size=(3, 3), activation='relu'))
model.add(Conv2D(384, kernel_size=(3, 3), activation='relu'))
model.add(Conv2D(384, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(3, 3), strides=(1, 1)))
model.add(BatchNormalization())

# dense layers:
model.add(Flatten())
model.add(Dense(4096, activation='tanh'))
model.add(Dense(4096, activation='tanh'))

# output layer:
model.add(Dense(10, activation='softmax'))
```



	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_alexnext_no_dropout/train	0.8839	0.8909	9	2.767 min
●	deep_network_alexnext_no_dropout/validation	0.8512	0.8475	9	2.767 min



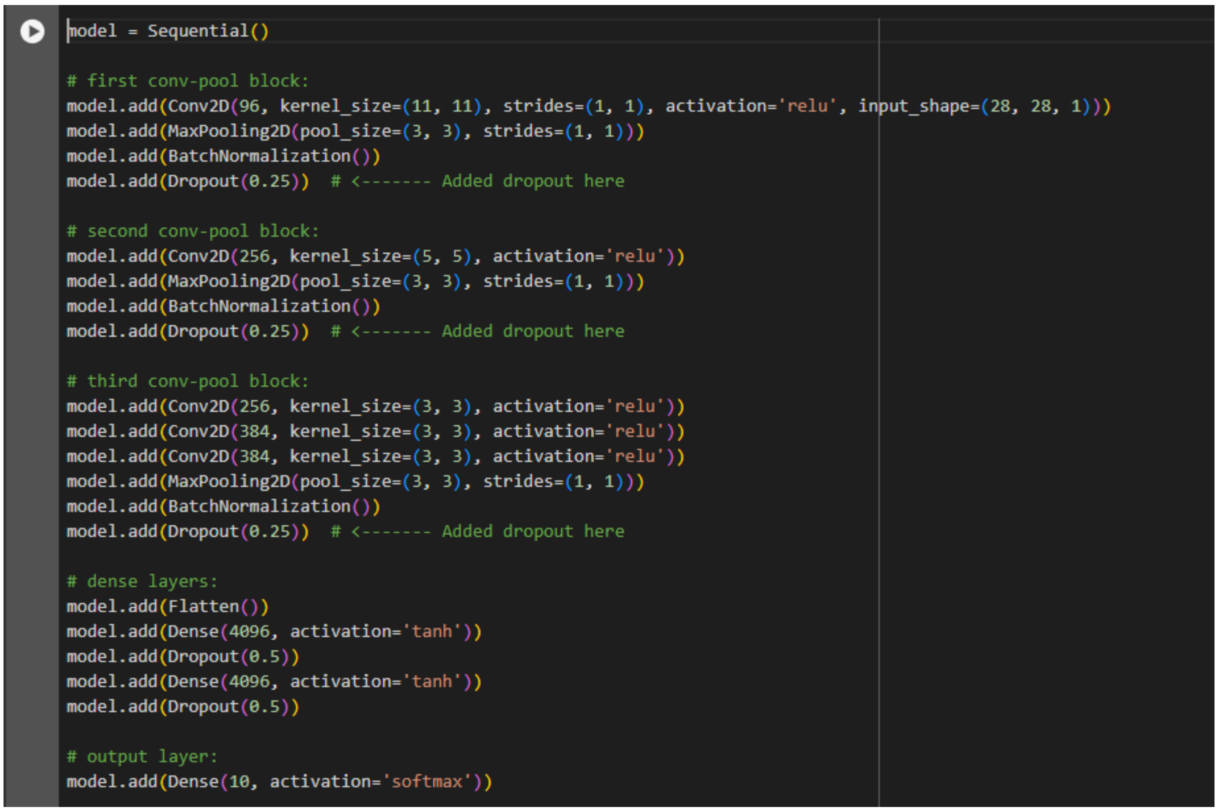
Run ↑	Smoothed	Value	Step	Relative
● deep_network_alexnext_no_dropout/train	0.3358	0.3158	9	2.767 min
● deep_network_alexnext_no_dropout/validation	0.4678	0.5213	9	2.767 min

Final Validation Accuracy Score: 0.8475

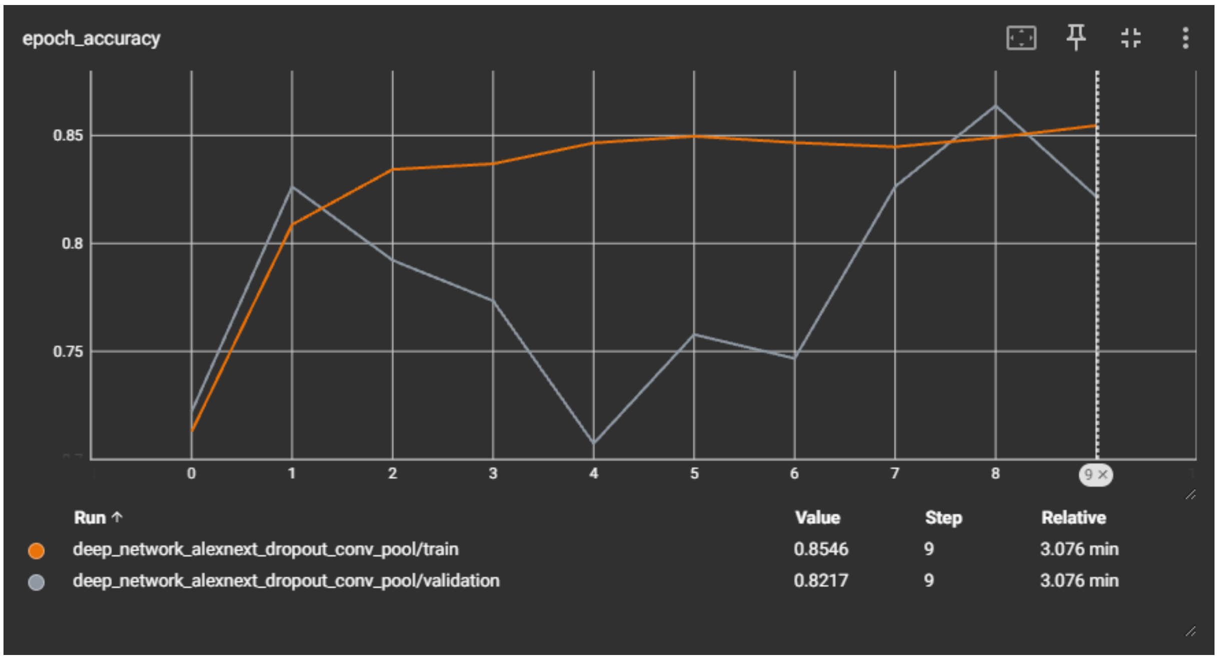
Comment:

After experimenting with the model architecture, we observed that removing dropout layers from the flatten layers resulted in a slight decrease in accuracy. Dropout layers are commonly used as a regularization technique to prevent overfitting by randomly dropping a fraction of the neurons during training. By including dropout layers after the flatten layers, we introduce regularization at the dense layers, helping to prevent the model from memorizing the training data and improving its generalization capability. The slight decrease in accuracy upon removing these dropout layers suggests that they were contributing to the model's ability to generalize to unseen data.
Additionally big difference between training accuracy and validation accuracy is disturbing.

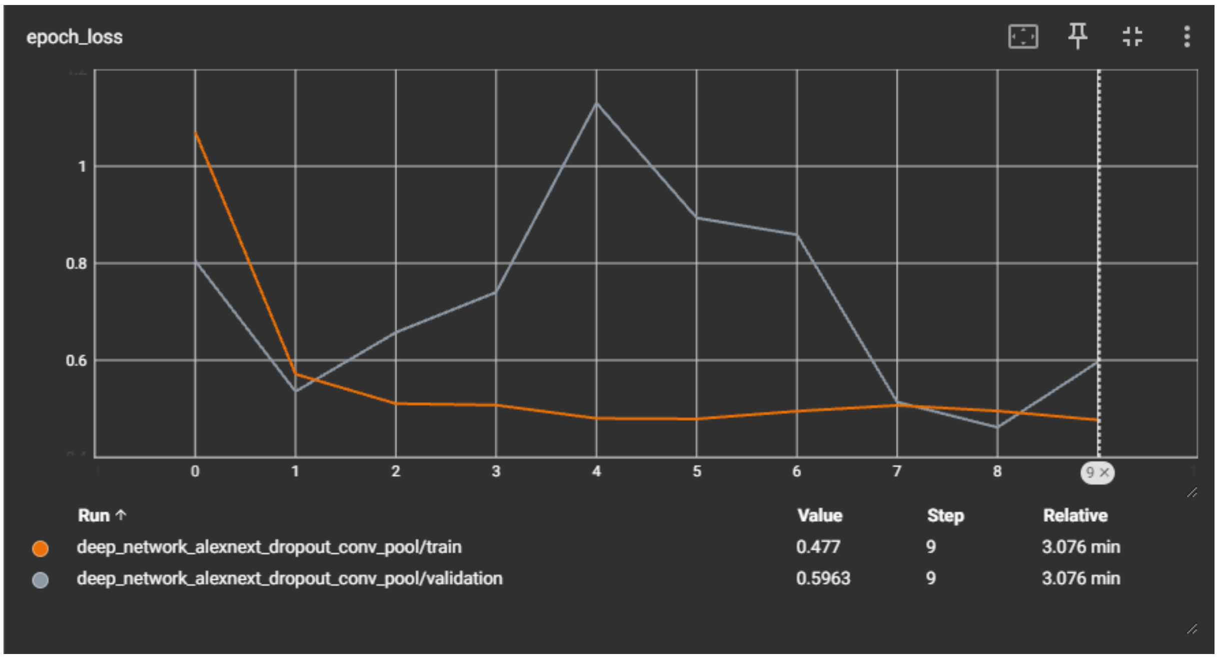
Added Dropouts after each Conv-Pool Block



```
model = Sequential()

# first conv-pool block:
model.add(Conv2D(96, kernel_size=(11, 11), strides=(1, 1), activation='relu', input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(3, 3), strides=(1, 1)))
model.add(BatchNormalization())
model.add(Dropout(0.25))  # <------- Added dropout here

# second conv-pool block:
model.add(Conv2D(256, kernel_size=(5, 5), activation='relu'))
model.add(MaxPooling2D(pool_size=(3, 3), strides=(1, 1)))
model.add(BatchNormalization())
model.add(Dropout(0.25))  # <------- Added dropout here

# third conv-pool block:
model.add(Conv2D(256, kernel_size=(3, 3), activation='relu'))
model.add(Conv2D(384, kernel_size=(3, 3), activation='relu'))
model.add(Conv2D(384, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(3, 3), strides=(1, 1)))
model.add(BatchNormalization())
model.add(Dropout(0.25))  # <------- Added dropout here

# dense layers:
model.add(Flatten())
model.add(Dense(4096, activation='tanh'))
model.add(Dropout(0.5))
model.add(Dense(4096, activation='tanh'))
model.add(Dropout(0.5))

# output layer:
model.add(Dense(10, activation='softmax'))
```



	Run ↑	Value	Step	Relative
●	deep_network_alexnext_dropout_conv_pool/train	0.8546	9	3.076 min
●	deep_network_alexnext_dropout_conv_pool/validation	0.8217	9	3.076 min



	Run ↑	Value	Step	Relative
●	deep_network_alexnext_dropout_conv_pool/train	0.477	9	3.076 min
●	deep_network_alexnext_dropout_conv_pool/validation	0.5963	9	3.076 min

Final Validation Accuracy Score: 0.8217 / 0.8638

Comment:

In analyzing the impact of adding extra dropout layers, we observe a nuanced effect on the model's convergence and accuracy. Despite a slower convergence rate initially, the model demonstrates promising signs of improvement, particularly evident in the higher accuracy achieved at the ninth epoch, surpassing previous iterations. However, this gain in accuracy is accompanied by a subsequent decrease in the final epoch, suggesting a potential trade-off between immediate convergence and long-term stability.

The phenomenon observed underscores the complex interplay between regularization techniques and model training dynamics. While the additional dropout layers contribute to regularization, promoting robustness and mitigating overfitting, their influence on convergence speed and final performance requires careful consideration. Balancing regularization strength with training efficiency is paramount in achieving optimal model performance.

In conclusion, while the inclusion of extra dropout layers may introduce variability in convergence behavior, it presents an avenue for enhancing model generalization and overcoming overfitting tendencies. Further experimentation and fine-tuning of hyperparameters may unveil strategies to harness the benefits of regularization while minimizing any adverse effects on convergence dynamics.

Regularizers

Regularizers are techniques used to add constraints on the weights of neural network models during training, with the aim of preventing overfitting and improving generalization performance. They work by penalizing large weights in the network, encouraging simpler models that are less prone to memorizing noise in the training data.

There are two main types of regularizers commonly used in neural networks:

1. **L1 Regularization (Lasso Regression):**

L1 regularization adds a penalty term to the loss function proportional to the absolute value of the weights. Mathematically, it adds the sum of the absolute values of the weights multiplied by a regularization parameter (lambda) to the loss function. This encourages sparsity in the weight matrix, leading to some weights being driven to exactly zero, effectively removing irrelevant features from the model.

L1 regularization can be expressed as:

$$\mathbf{L_1} = \boldsymbol{\lambda} \cdot \left|\left|\vec{\boldsymbol{w}}\right|\right|\square\mathbf{1}$$

where$\left|\left|\vec{\boldsymbol{w}}\right|\right|_{\mathbf{1}}$ is the L1 norm of the weight vector.

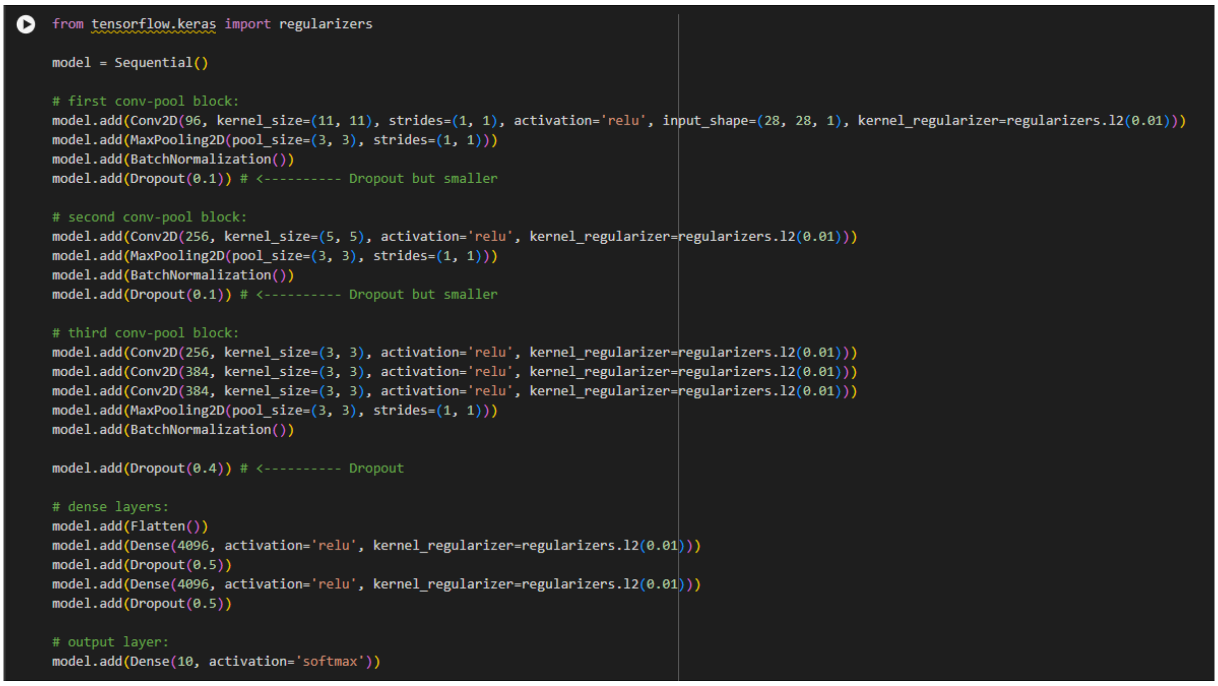
2. **L2 Regularization (Ridge Regression):**

L2 regularization adds a penalty term to the loss function proportional to the squared magnitude of the weights. Mathematically, it adds the sum of the squared values of the weights multiplied by a regularization parameter (lambda) to the loss function. This encourages smaller weights in the network, preventing any single weight from becoming too large and dominating the learning process.

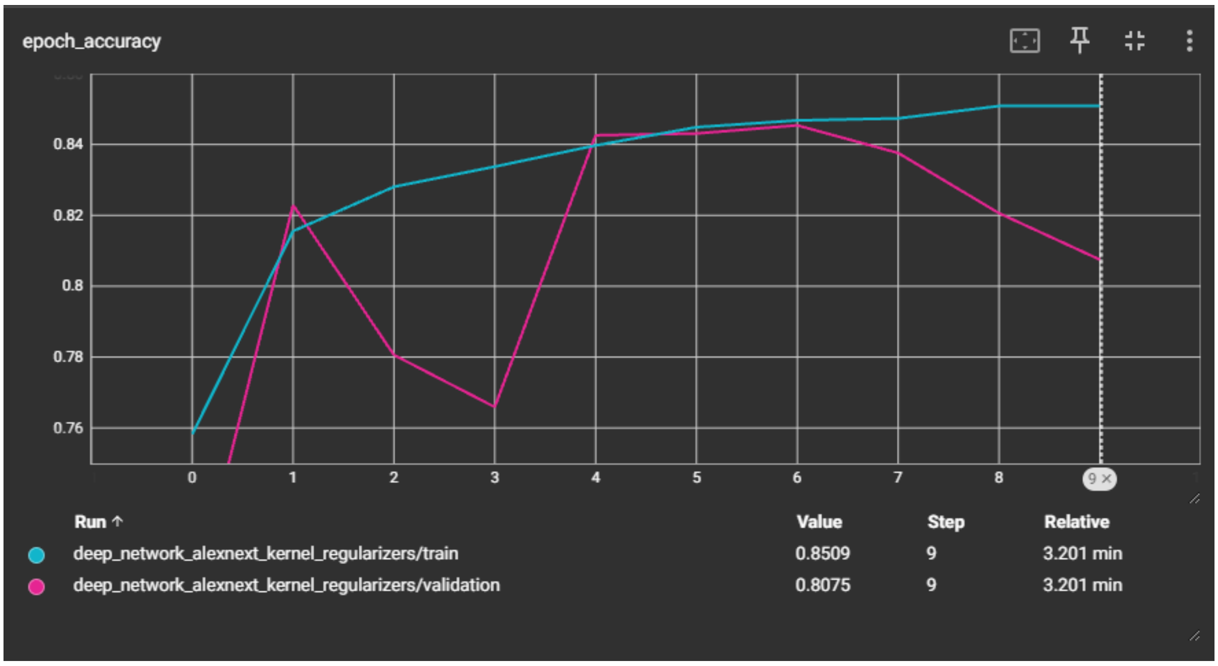
L2 regularization can be expressed as:

$$L_2 = \lambda \cdot \left|\left|\vec{w}\right|\right|_2^2$$

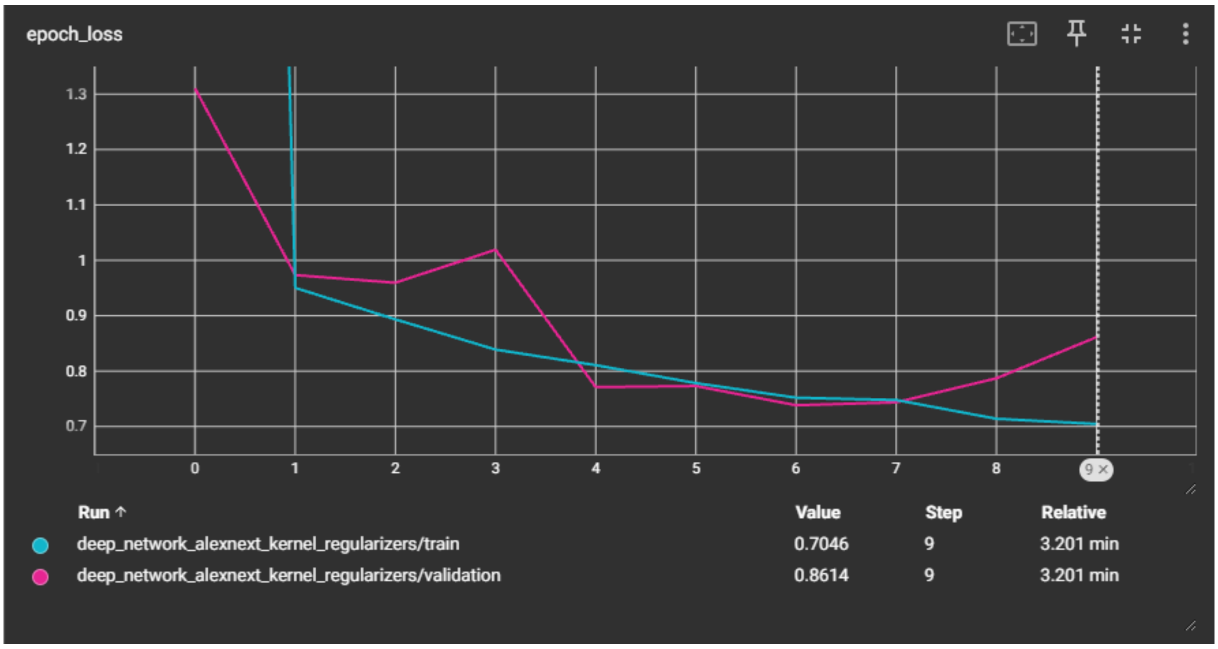
Regularization helps to control the complexity of the model by discouraging large weights, which can lead to overfitting. By applying regularization, the model is encouraged to learn simpler patterns that are more likely to generalize well to unseen data. The regularization parameter (lambda) controls the strength of regularization, with larger values of lambda imposing stronger penalties on the weights. Choosing an appropriate value for lambda is often done through hyperparameter tuning.



```python
from tensorflow.keras import regularizers

model = Sequential()

# first conv-pool block:
model.add(Conv2D(96, kernel_size=(11, 11), strides=(1, 1), activation='relu', input_shape=(28, 28, 1), kernel_regularizer=regularizers.l2(0.01)))
model.add(MaxPooling2D(pool_size=(3, 3), strides=(1, 1)))
model.add(BatchNormalization())
model.add(Dropout(0.1)) # <---------- Dropout but smaller

# second conv-pool block:
model.add(Conv2D(256, kernel_size=(5, 5), activation='relu', kernel_regularizer=regularizers.l2(0.01)))
model.add(MaxPooling2D(pool_size=(3, 3), strides=(1, 1)))
model.add(BatchNormalization())
model.add(Dropout(0.1)) # <---------- Dropout but smaller

# third conv-pool block:
model.add(Conv2D(256, kernel_size=(3, 3), activation='relu', kernel_regularizer=regularizers.l2(0.01)))
model.add(Conv2D(384, kernel_size=(3, 3), activation='relu', kernel_regularizer=regularizers.l2(0.01)))
model.add(Conv2D(384, kernel_size=(3, 3), activation='relu', kernel_regularizer=regularizers.l2(0.01)))
model.add(MaxPooling2D(pool_size=(3, 3), strides=(1, 1)))
model.add(BatchNormalization())

model.add(Dropout(0.4)) # <---------- Dropout

# dense layers:
model.add(Flatten())
model.add(Dense(4096, activation='relu', kernel_regularizer=regularizers.l2(0.01)))
model.add(Dropout(0.5))
model.add(Dense(4096, activation='relu', kernel_regularizer=regularizers.l2(0.01)))
model.add(Dropout(0.5))

# output layer:
model.add(Dense(10, activation='softmax'))
```



	Run ↑	Value	Step	Relative
●	deep_network_alexnext_kernel_regularizers/train	0.8509	9	3.201 min
●	deep_network_alexnext_kernel_regularizers/validation	0.8075	9	3.201 min



Run ↑	Value	Step	Relative
● deep_network_alexnext_kernel_regularizers/train	0.7046	9	3.201 min
● deep_network_alexnext_kernel_regularizers/validation	0.8614	9	3.201 min

Final Validation Accuracy Score: 0.8075 / 0.8454

It seems that the regularizers didn't have the intended effect on improving model performance, as evidenced by the observed loss and accuracy trends during training and validation.

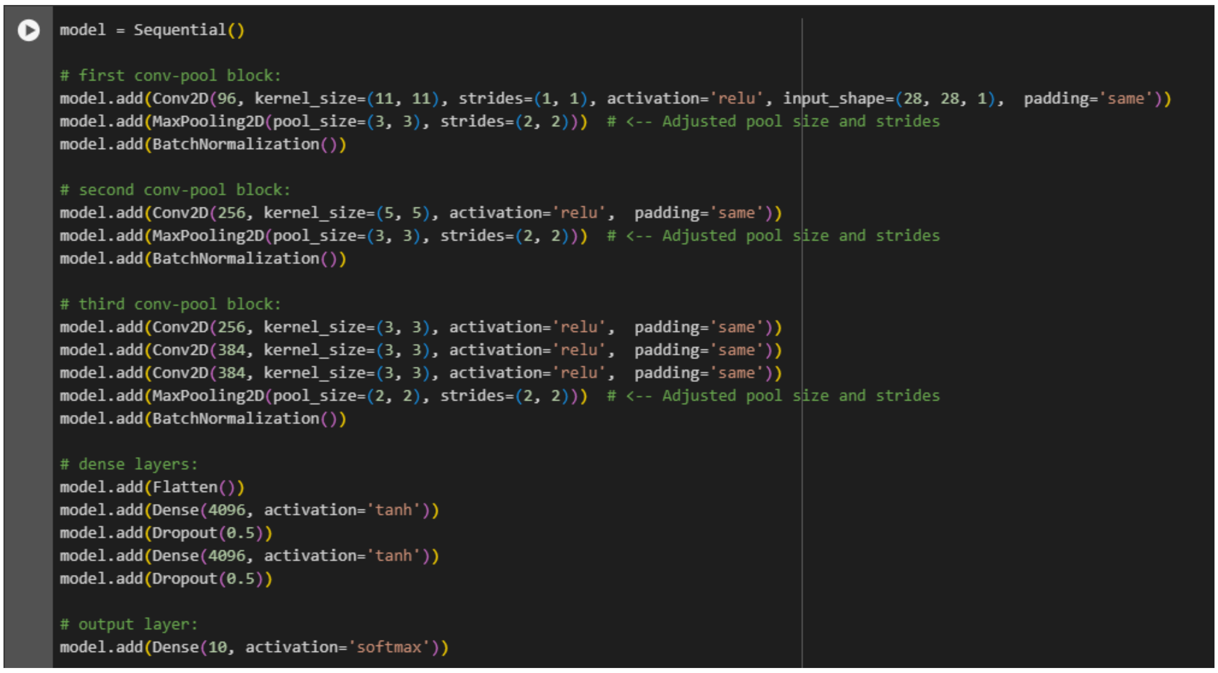
While regularizers such as L1 and L2 are typically effective in preventing overfitting by penalizing large weights, their impact may be overshadowed or complemented by other factors in the model architecture or training process.

In this case, considering the model's loss and accuracy trajectories, it's possible that the dropout layers had a more dominant influence on model regularization compared to the regularizers. Dropout layers randomly deactivate a fraction of neurons during training, effectively introducing noise and preventing co-adaptation of features, which can help prevent overfitting.

Additionally, the effect of regularization techniques can be highly dependent on factors such as the complexity of the dataset, the architecture of the model, and the choice of hyperparameters. It's essential to carefully tune and experiment with different regularization techniques and hyperparameters to find the optimal configuration for the specific task and dataset.

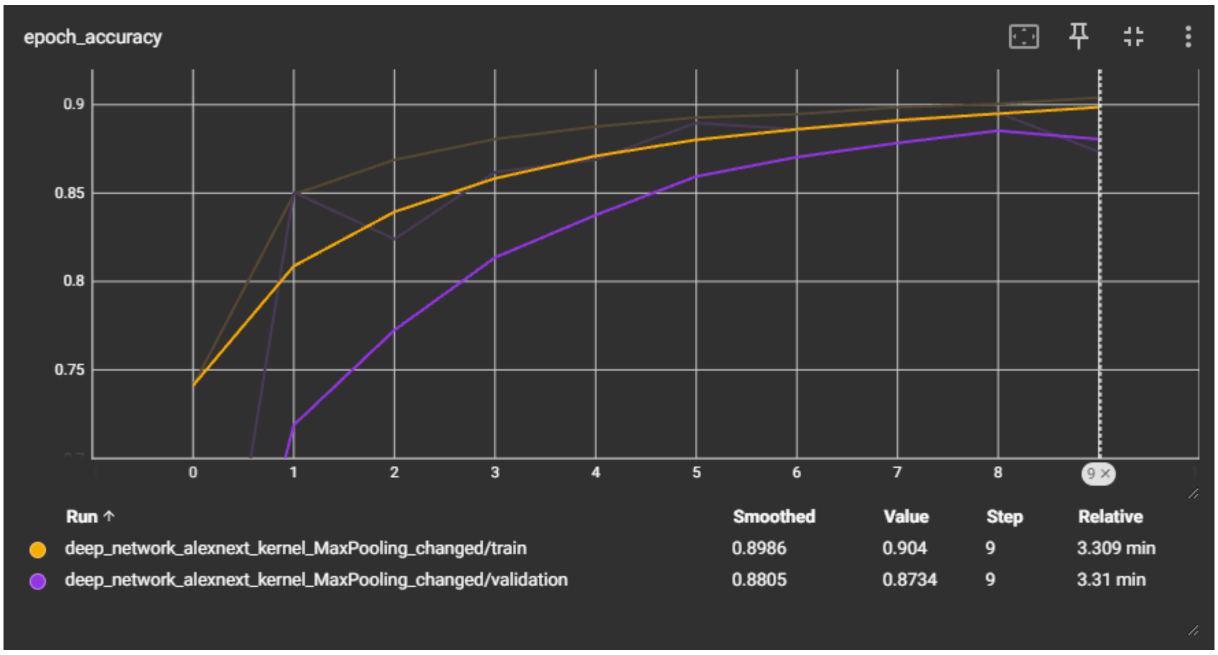
Therefore, while regularizers may still contribute to model regularization, further experimentation and analysis are warranted to determine their relative effectiveness compared to dropout layers and other regularization techniques in this particular scenario.

Max Pool 2D Modified – Strides and pool size

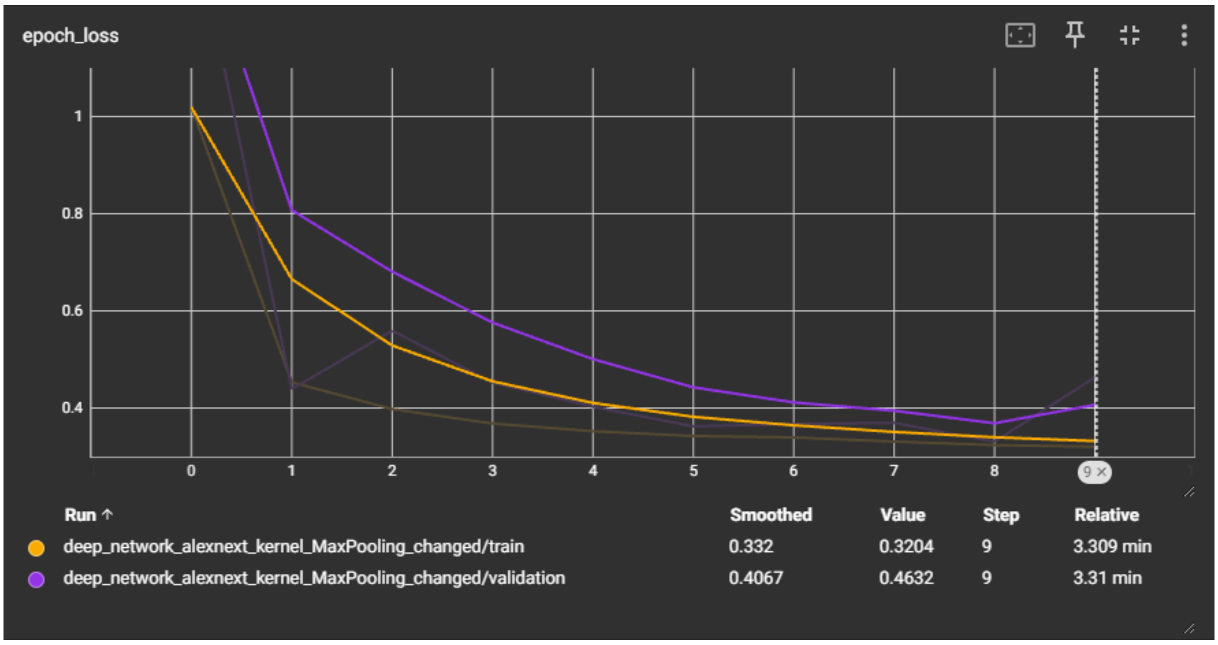


```
model = Sequential()

# first conv-pool block:
model.add(Conv2D(96, kernel_size=(11, 11), strides=(1, 1), activation='relu', input_shape=(28, 28, 1),  padding='same'))
model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))  # <-- Adjusted pool size and strides
model.add(BatchNormalization())

# second conv-pool block:
model.add(Conv2D(256, kernel_size=(5, 5), activation='relu',  padding='same'))
model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))  # <-- Adjusted pool size and strides
model.add(BatchNormalization())

# third conv-pool block:
model.add(Conv2D(256, kernel_size=(3, 3), activation='relu',  padding='same'))
model.add(Conv2D(384, kernel_size=(3, 3), activation='relu',  padding='same'))
model.add(Conv2D(384, kernel_size=(3, 3), activation='relu',  padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))  # <-- Adjusted pool size and strides
model.add(BatchNormalization())

# dense layers:
model.add(Flatten())
model.add(Dense(4096, activation='tanh'))
model.add(Dropout(0.5))
model.add(Dense(4096, activation='tanh'))
model.add(Dropout(0.5))

# output layer:
model.add(Dense(10, activation='softmax'))
```

Adding padding="same" to the convolutional layers ensures that the output dimensions remain valid after the downsampling operation.



	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_alexnext_kernel_MaxPooling_changed/train	0.8986	0.904	9	3.309 min
●	deep_network_alexnext_kernel_MaxPooling_changed/validation	0.8805	0.8734	9	3.31 min



Run ↑	Smoothed	Value	Step	Relative
● deep_network_alexnext_kernel_MaxPooling_changed/train	0.332	0.3204	9	3.309 min
● deep_network_alexnext_kernel_MaxPooling_changed/validation	0.4067	0.4632	9	3.31 min

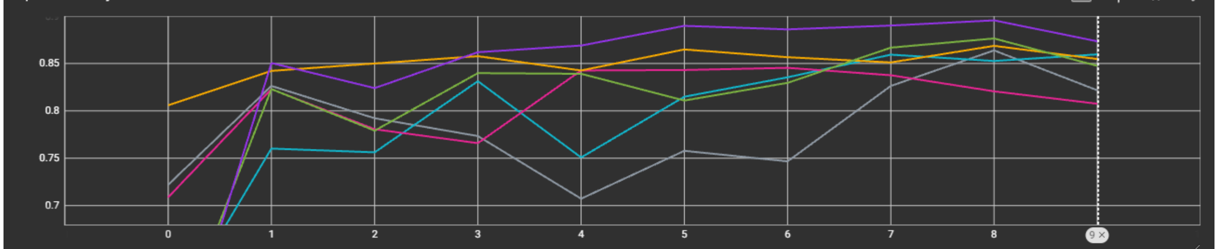
Final Validation Accuracy Score: 0.8734 / **0.8955**

The training started with a significant gap between the training and validation accuracies, indicating some degree of overfitting. However, as the training progressed, the model's performance improved steadily. By the end of the training, the validation accuracy reached one of the highest points observed so far, almost reaching 90%. This suggests that the model was able to generalize well to unseen data.

It's worth noting that towards the end of the training (in the last epoch), there was a slight decrease in the validation accuracy, which might indicate a slight overfitting tendency or a convergence issue. Further analysis, such as tuning hyperparameters or adjusting the model architecture, could be explored to address this and potentially improve the model's performance. Overall, the model shows promise, with validation accuracy consistently above 85% throughout the training process.

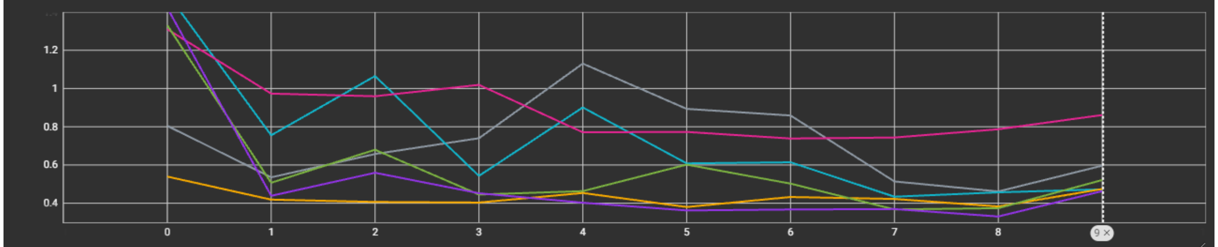
AlexNet Comparation

epoch_accuracy



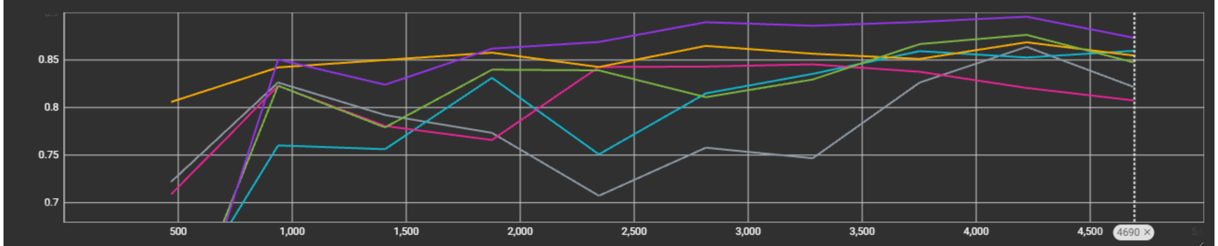
	Run ↑	Value	Step	Relative
●	deep_network_alexnext_default/validation	0.8596	9	2.754 min
●	deep_network_alexnext_dropout_conv_pool/validation	0.8217	9	3.076 min
●	deep_network_alexnext_kernel_MaxPooling_changed/validation	0.8734	9	3.31 min
●	deep_network_alexnext_kernel_regularizers/validation	0.8075	9	3.201 min
●	deep_network_alexnext_no_batch/validation	0.8547	9	2.561 min
●	deep_network_alexnext_no_dropout/validation	0.8475	9	2.767 min

epoch_loss

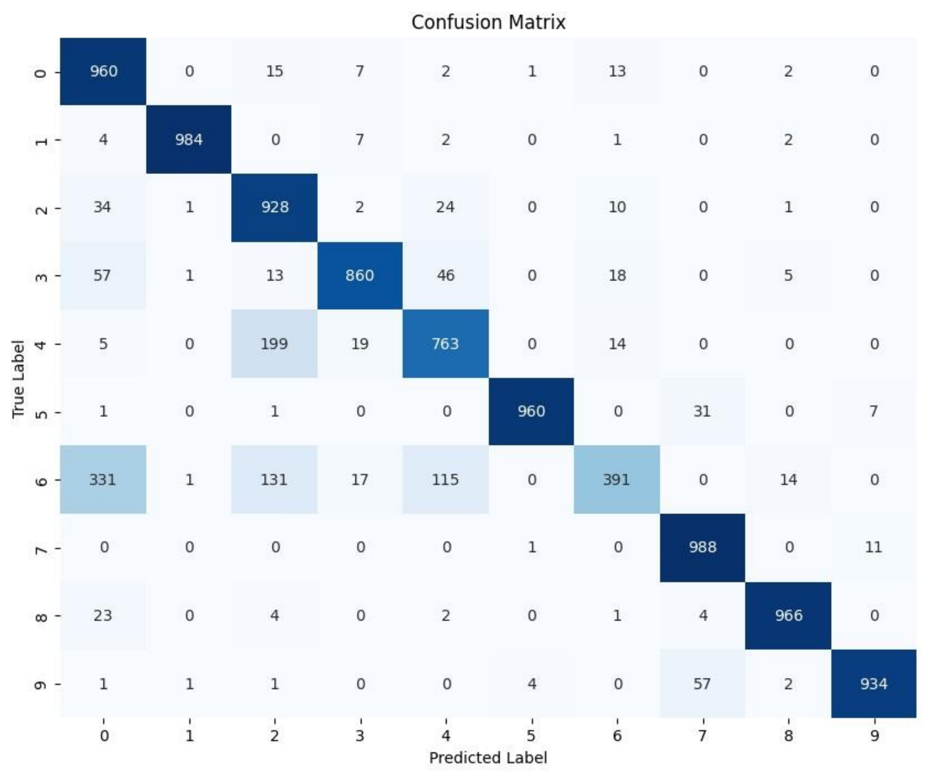


	Run ↑	Value	Step	Relative
●	deep_network_alexnext_default/validation	0.4726	9	2.754 min
●	deep_network_alexnext_dropout_conv_pool/validation	0.5963	9	3.076 min
●	deep_network_alexnext_kernel_MaxPooling_changed/validation	0.4632	9	3.31 min
●	deep_network_alexnext_kernel_regularizers/validation	0.8614	9	3.201 min
●	deep_network_alexnext_no_batch/validation	0.4743	9	2.561 min
●	deep_network_alexnext_no_dropout/validation	0.5213	9	2.767 min

evaluation_accuracy_vs_iterations



	Run ↑	Value	Step	Relative
●	deep_network_alexnext_default/validation	0.8596	4,690	2.749 min
●	deep_network_alexnext_dropout_conv_pool/validation	0.8217	4,690	3.085 min
●	deep_network_alexnext_kernel_MaxPooling_changed/validation	0.8734	4,690	3.303 min
●	deep_network_alexnext_kernel_regularizers/validation	0.8075	4,690	3.2 min
●	deep_network_alexnext_no_batch/validation	0.8547	4,690	2.573 min
●	deep_network_alexnext_no_dropout/validation	0.8475	4,690	2.768 min



Confusion Matrix

True Label \ Predicted Label	0	1	2	3	4	5	6	7	8	9
0	960	0	15	7	2	1	13	0	2	0
1	4	984	0	7	2	0	1	0	2	0
2	34	1	928	2	24	0	10	0	1	0
3	57	1	13	860	46	0	18	0	5	0
4	5	0	199	19	763	0	14	0	0	0
5	1	0	1	0	0	960	0	31	0	7
6	331	1	131	17	115	0	391	0	14	0
7	0	0	0	0	0	1	0	988	0	11
8	23	0	4	0	2	0	1	4	966	0
9	1	1	1	0	0	4	0	57	2	934

The confusion matrix provides a comprehensive view of the model's performance by showcasing the counts of true positive, true negative, false positive, and false negative predictions for each class in Fashion MNIST dataset.

Looking at the matrix:

- The diagonal elements represent the number of correctly classified samples for each class.
- Off-diagonal elements indicate misclassifications between different classes.

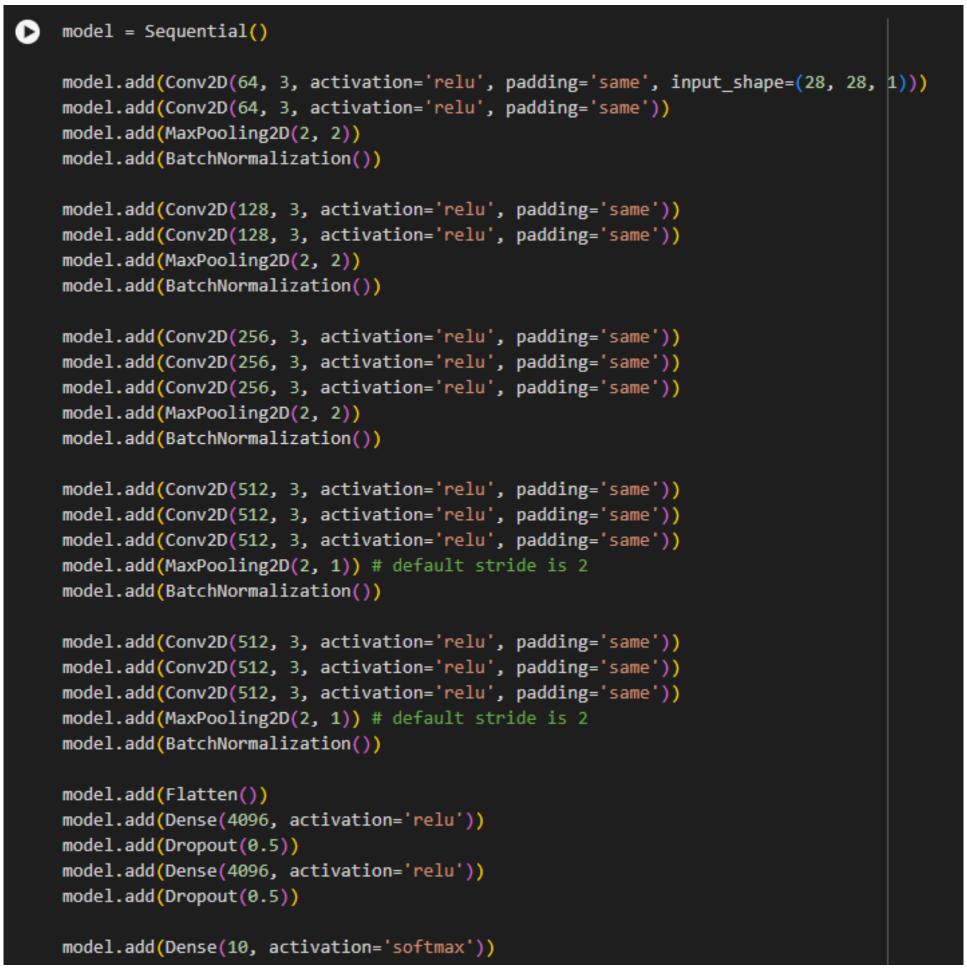
Observations:

- Class 0 (T-shirt/top) has a high number of true positives (960), but it is often confused with class 6 (Shirt), with 331 misclassifications.
- Class 1 (Trousers) has high accuracy with 984 true positives, but there are a few misclassifications with class 3 (Dress) and class 4 (Coat).
- Class 2 (Pullover) has a significant number of misclassifications with class 4 (Coat) and class 6 (Shirt).
- Class 3 (Dress) is frequently confused with class 0 (T-shirt/top) and class 4 (Coat).
- Class 4 (Coat) has considerable misclassifications with class 2 (Pullover), class 3 (Dress), and class 6 (Shirt).

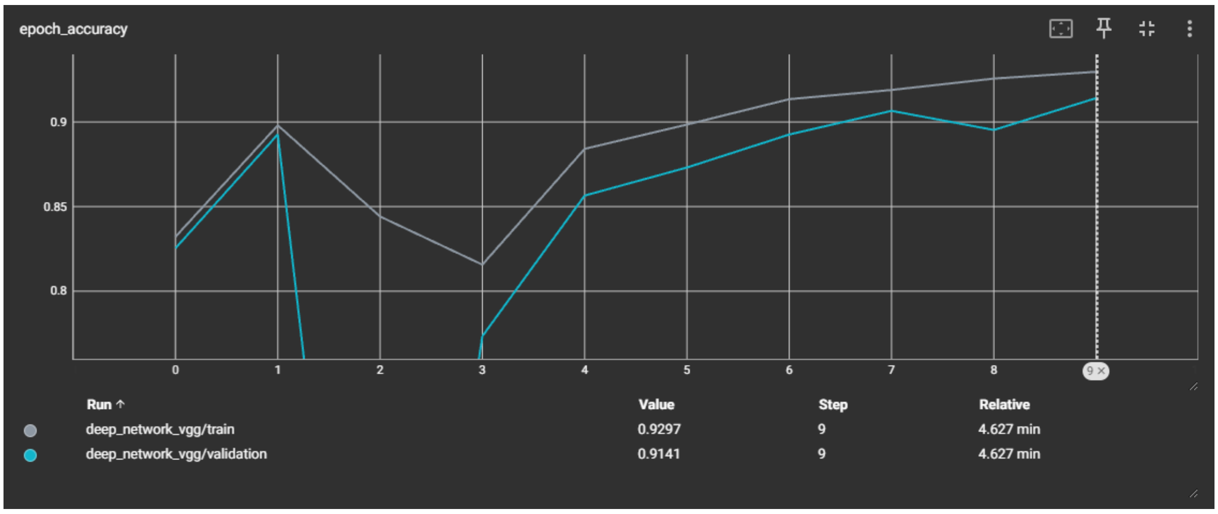
- Class 5 (Sandal) has a few misclassifications with class 7 (Sneaker) and class 9 (Ankle boot).
- Class 6 (Shirt) shows misclassifications with various classes, especially class 0 (T-shirt/top) and class 2 (Pullover).
- Class 7 (Sneaker) is generally classified correctly but has a few misclassifications with class 9 (Ankle boot).
- Class 8 (Bag) has a high number of true positives with minimal misclassifications.
- Class 9 (Ankle boot) is often confused with class 7 (Sneaker), with 57 misclassifications.

Overall, while the model demonstrates high accuracy in some classes, there are certain classes that exhibit more confusion with others. Further analysis and model refinement may help improve performance, particularly in distinguishing between similar classes.

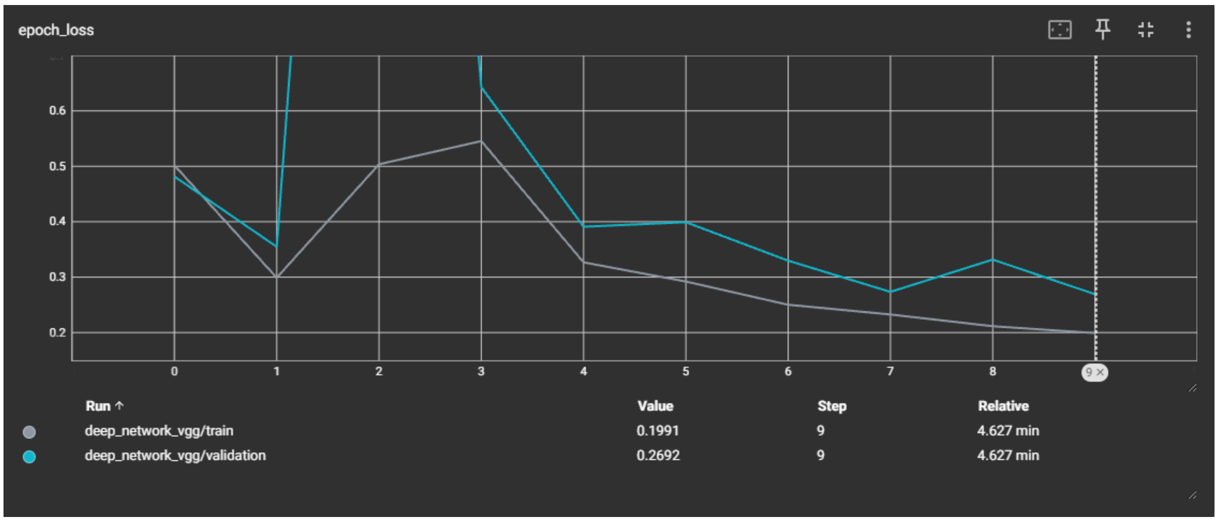
VGG on Fashion MNIST



```
model = Sequential()

model.add(Conv2D(64, 3, activation='relu', padding='same', input_shape=(28, 28, 1)))
model.add(Conv2D(64, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 2))
model.add(BatchNormalization())

model.add(Conv2D(128, 3, activation='relu', padding='same'))
model.add(Conv2D(128, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 2))
model.add(BatchNormalization())

model.add(Conv2D(256, 3, activation='relu', padding='same'))
model.add(Conv2D(256, 3, activation='relu', padding='same'))
model.add(Conv2D(256, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 2))
model.add(BatchNormalization())

model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 1)) # default stride is 2
model.add(BatchNormalization())

model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 1)) # default stride is 2
model.add(BatchNormalization())

model.add(Flatten())
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))

model.add(Dense(10, activation='softmax'))
```



Run ↑	Value	Step	Relative
● deep_network_vgg/train	0.9297	9	4.627 min
● deep_network_vgg/validation	0.9141	9	4.627 min



Run ↑	Value	Step	Relative
● deep_network_vgg/train	0.1991	9	4.627 min
● deep_network_vgg/validation	0.2692	9	4.627 min

Final Validation Accuracy Score: 0.9141

The VGG model demonstrates varying performance across epochs, with fluctuations in both training and validation accuracy.

Observations:

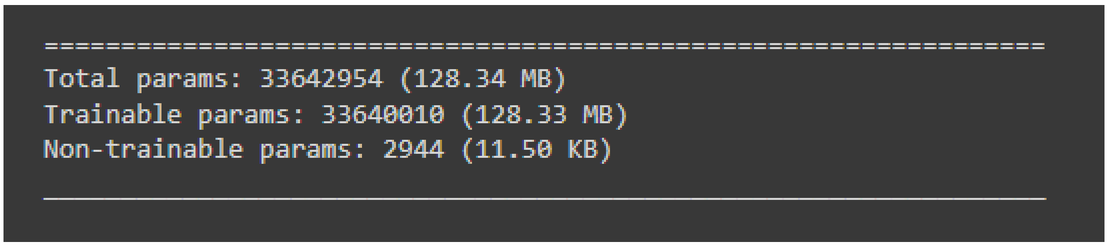
- Epochs 1 and 2 show promising results, with the model achieving high accuracy on both the training and validation sets. The validation accuracy increases steadily, indicating effective learning and generalization.

- In Epoch 3, there is a significant drop in validation accuracy, suggesting that the model may be overfitting to the training data or encountering difficulties in generalizing to unseen examples. The high loss value further supports this observation.

- Epochs 4 and 5 show improvements in both training and validation accuracy, indicating that the model is recovering from the performance dip observed in Epoch 3. However, the validation accuracy remains slightly lower than in the initial epochs.

- The training and validation accuracies continue to improve in Epochs 6 to 10, with the model achieving its highest validation accuracy of 91.41% in the final epoch. This suggests that the model's performance stabilizes and continues to improve with further training.

Overall, while the VGG model demonstrates strong performance on the Fashion MNIST dataset, there are fluctuations in accuracy across epochs, indicating potential challenges in training and generalization. Further analysis and fine-tuning of the model parameters may help stabilize and improve its performance over time.



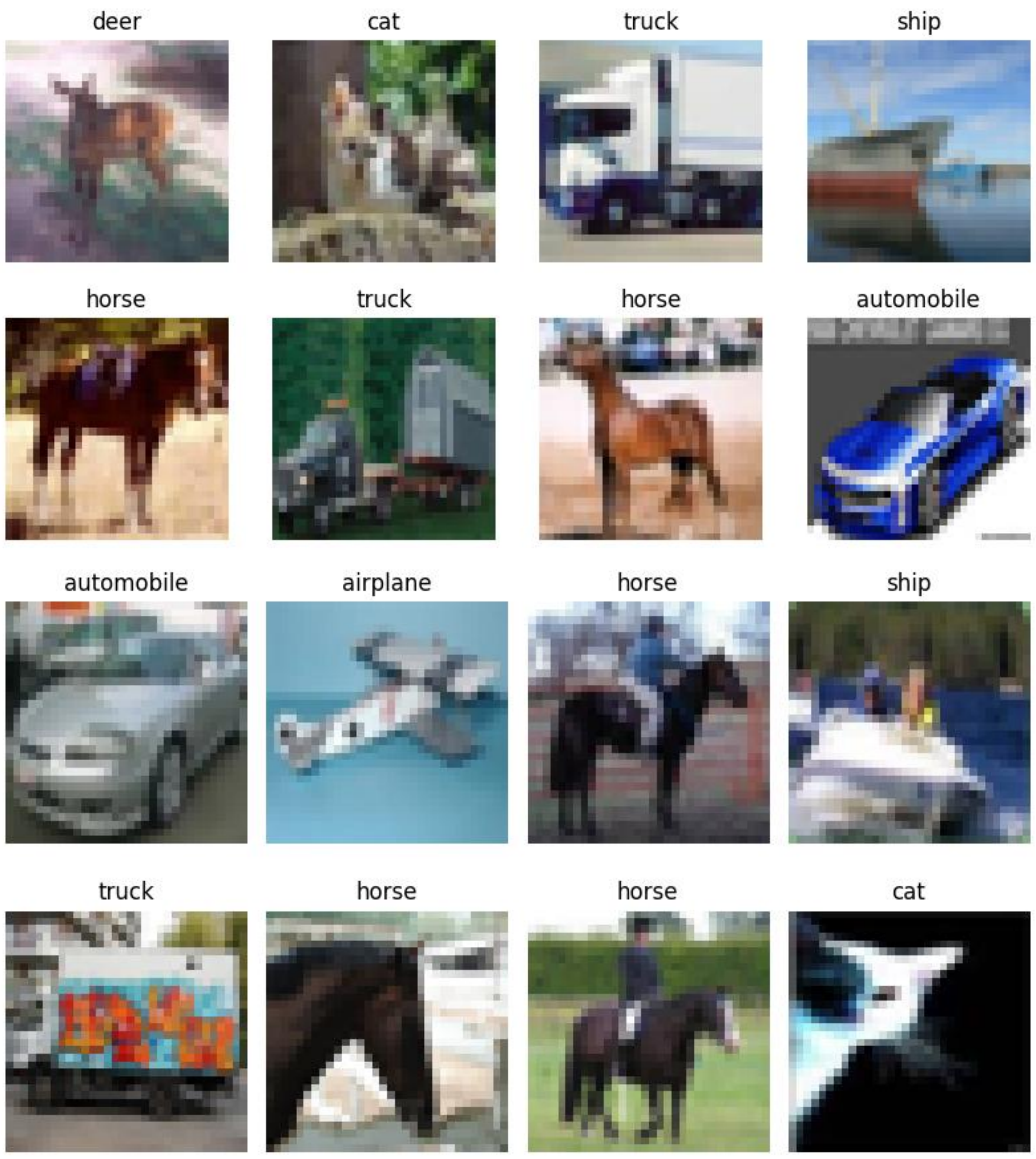
```
=================================================================
Total params: 33642954 (128.34 MB)
Trainable params: 33640010 (128.33 MB)
Non-trainable params: 2944 (11.50 KB)
_________________________________________________________________
```

CIFAR-10 Dataset

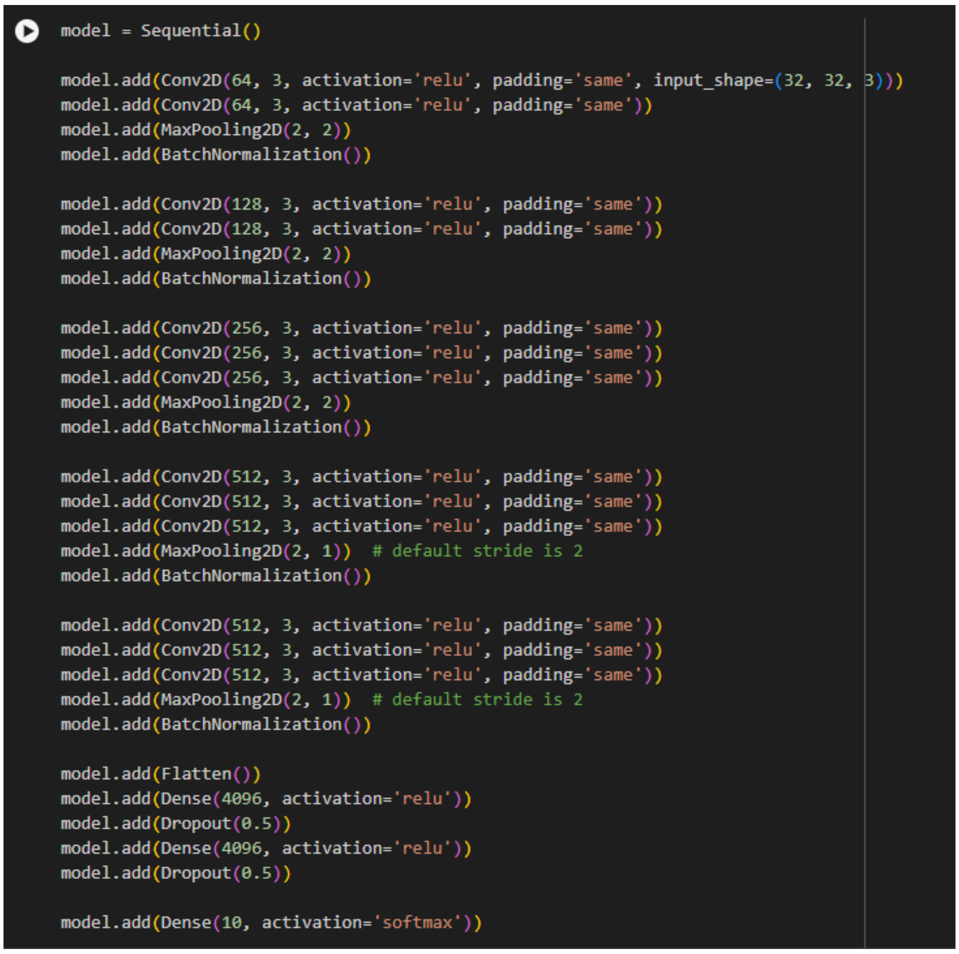
The CIFAR-10 dataset is a widely used benchmark dataset in the field of computer vision. It consists of 60,000 32x32 color images in 10 classes, with 6,000 images per class. The dataset is divided into a training set of 50,000 images and a test set of 10,000 images. Each image is labeled with one of the following classes:

Label	Category
0	Airplane
1	Automobile
2	Bird
3	Cat
4	Deer
5	Dog
6	Frog
7	Horse
8	Ship
9	Truck

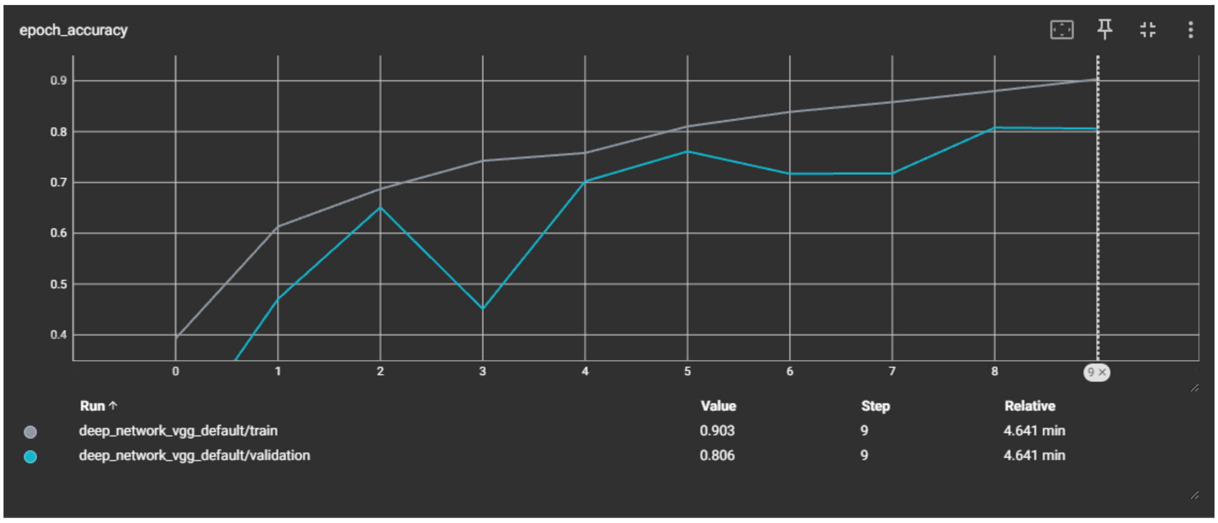
The images in CIFAR-10 are of size 32x32 pixels with three color channels (RGB), resulting in a total of 3,072 features per image.


deer
cat
truck
ship
horse
truck
horse
automobile
automobile
airplane
horse
ship
truck
horse
horse
cat

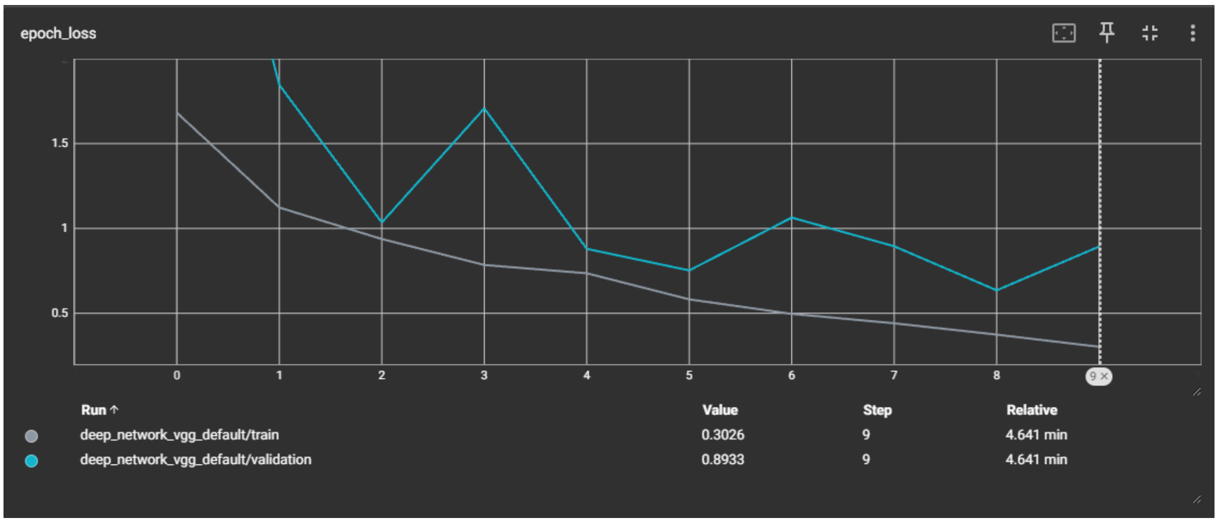
Default VGG Net vs CIFAR Dataset



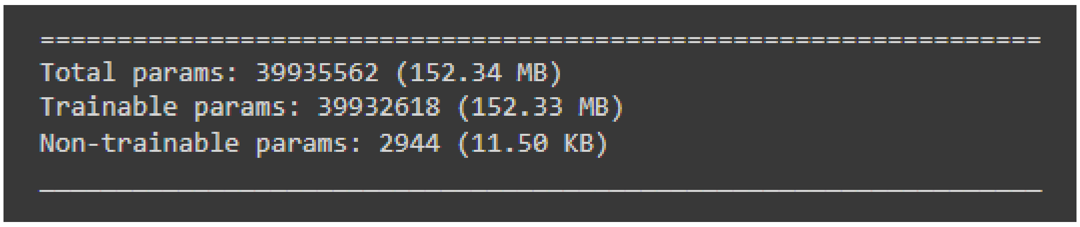
```
model = Sequential()

model.add(Conv2D(64, 3, activation='relu', padding='same', input_shape=(32, 32, 3)))
model.add(Conv2D(64, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 2))
model.add(BatchNormalization())

model.add(Conv2D(128, 3, activation='relu', padding='same'))
model.add(Conv2D(128, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 2))
model.add(BatchNormalization())

model.add(Conv2D(256, 3, activation='relu', padding='same'))
model.add(Conv2D(256, 3, activation='relu', padding='same'))
model.add(Conv2D(256, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 2))
model.add(BatchNormalization())

model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 1))  # default stride is 2
model.add(BatchNormalization())

model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 1))  # default stride is 2
model.add(BatchNormalization())

model.add(Flatten())
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))

model.add(Dense(10, activation='softmax'))
```



	Run ↑	Value	Step	Relative
●	deep_network_vgg_default/train	0.903	9	4.641 min
●	deep_network_vgg_default/validation	0.806	9	4.641 min



	Run ↑	Value	Step	Relative
●	deep_network_vgg_default/train	0.3026	9	4.641 min
●	deep_network_vgg_default/validation	0.8933	9	4.641 min

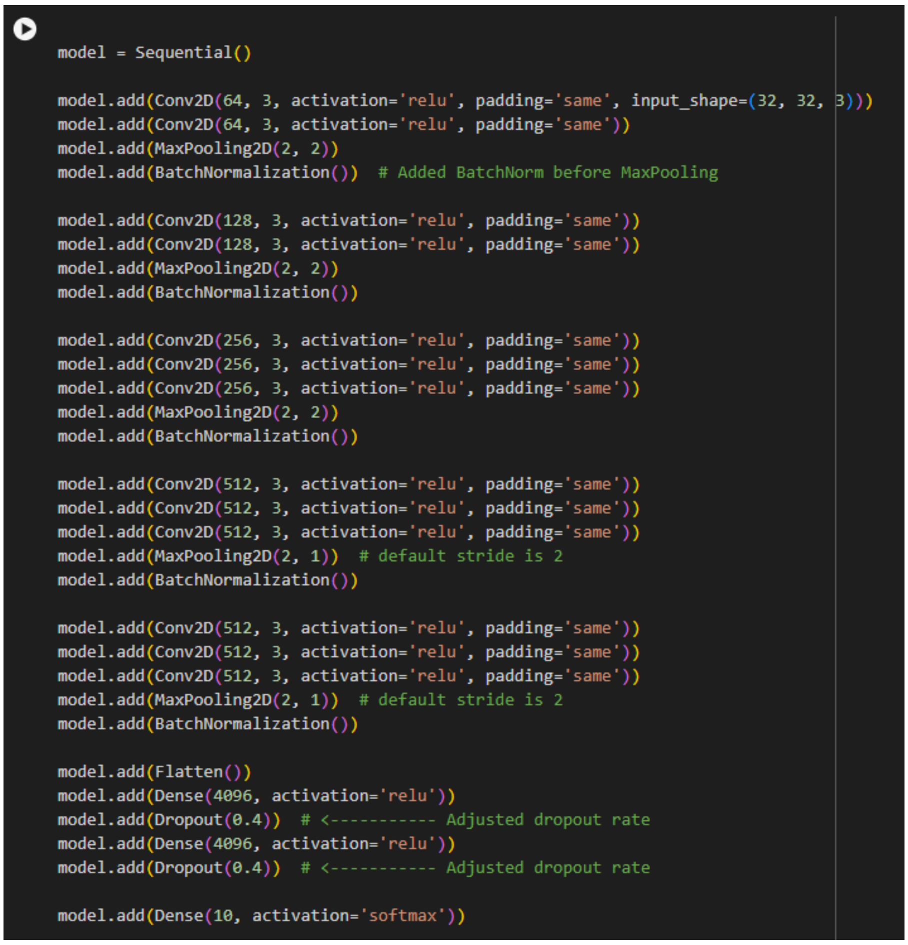


```
=================================================================
Total params: 39935562 (152.34 MB)
Trainable params: 39932618 (152.33 MB)
Non-trainable params: 2944 (11.50 KB)
_________________________________________________________________
```

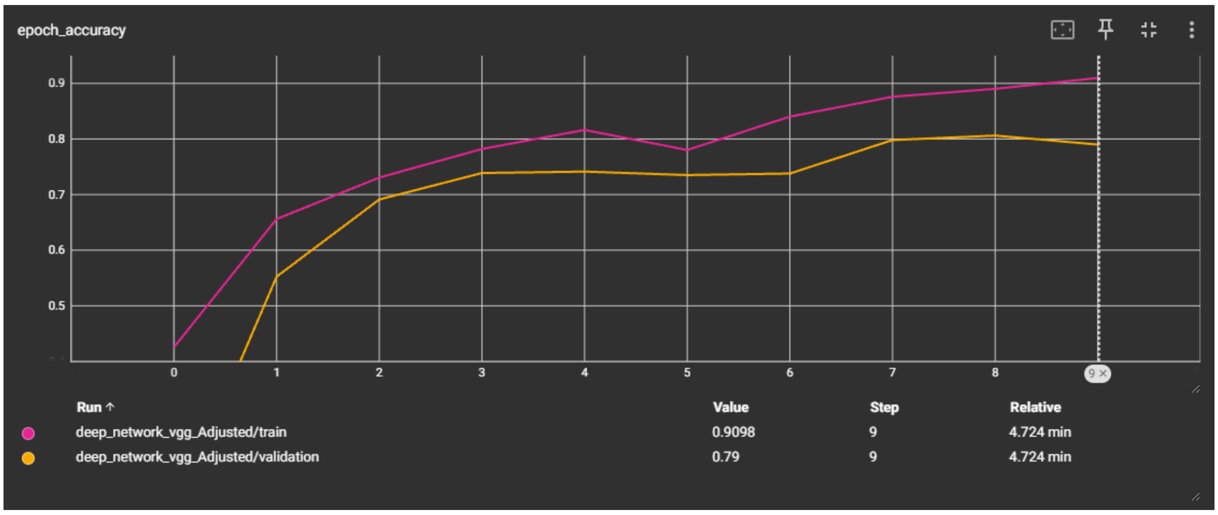
Final Validation Accuracy Score: 0.8060

VGG Net: Adjusted BatchNorm and Dropout

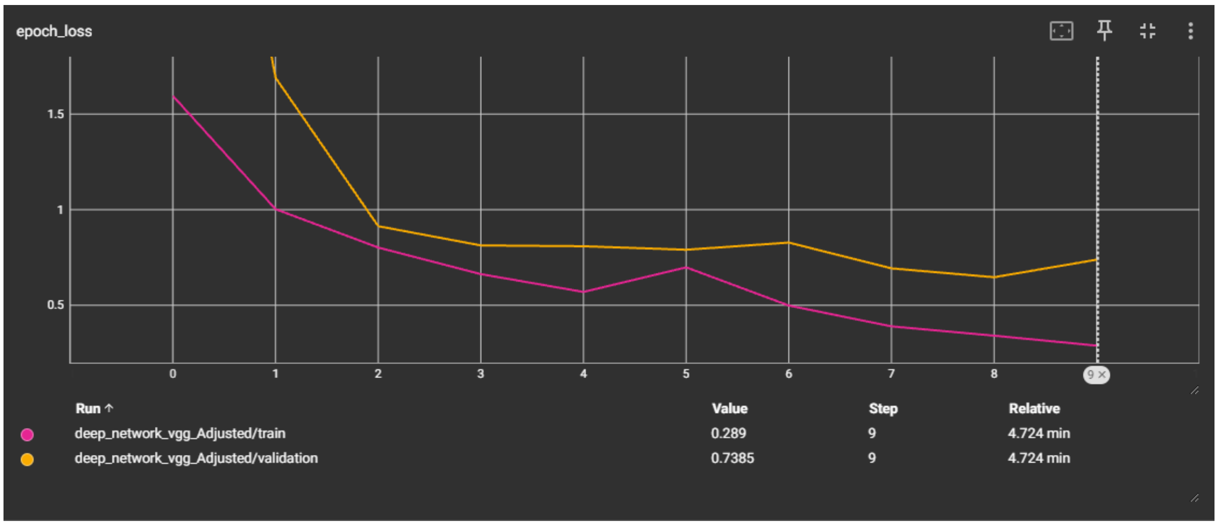
Now after each block there is batch norm. Dropout is slightly smaller.



```
model = Sequential()

model.add(Conv2D(64, 3, activation='relu', padding='same', input_shape=(32, 32, 3)))
model.add(Conv2D(64, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 2))
model.add(BatchNormalization())  # Added BatchNorm before MaxPooling

model.add(Conv2D(128, 3, activation='relu', padding='same'))
model.add(Conv2D(128, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 2))
model.add(BatchNormalization())

model.add(Conv2D(256, 3, activation='relu', padding='same'))
model.add(Conv2D(256, 3, activation='relu', padding='same'))
model.add(Conv2D(256, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 2))
model.add(BatchNormalization())

model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 1))  # default stride is 2
model.add(BatchNormalization())

model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 1))  # default stride is 2
model.add(BatchNormalization())

model.add(Flatten())
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.4))  # <---------- Adjusted dropout rate
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.4))  # <---------- Adjusted dropout rate

model.add(Dense(10, activation='softmax'))
```



	Run ↑	Value	Step	Relative
●	deep_network_vgg_Adjusted/train	0.9098	9	4.724 min
●	deep_network_vgg_Adjusted/validation	0.79	9	4.724 min



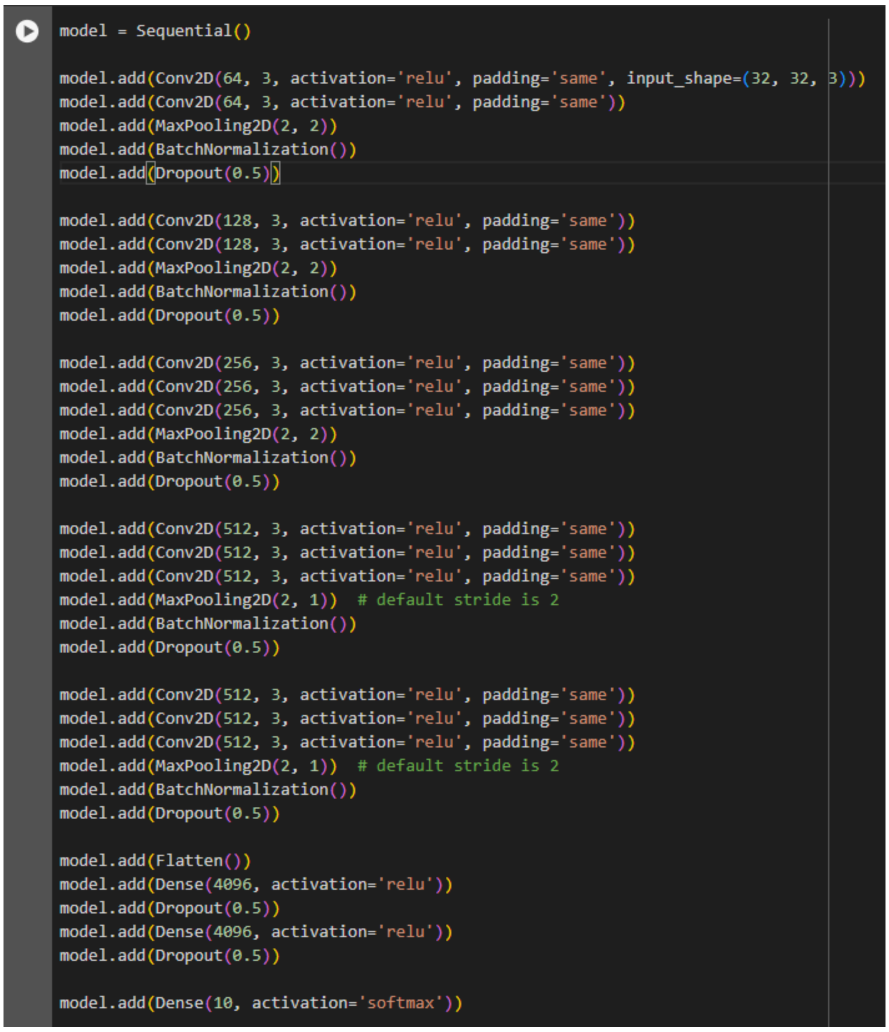
Run ↑	Value	Step	Relative
● deep_network_vgg_Adjusted/train	0.289	9	4.724 min
● deep_network_vgg_Adjusted/validation	0.7385	9	4.724 min

Final Validation Accuracy Score: 0.7900 / 0.8062

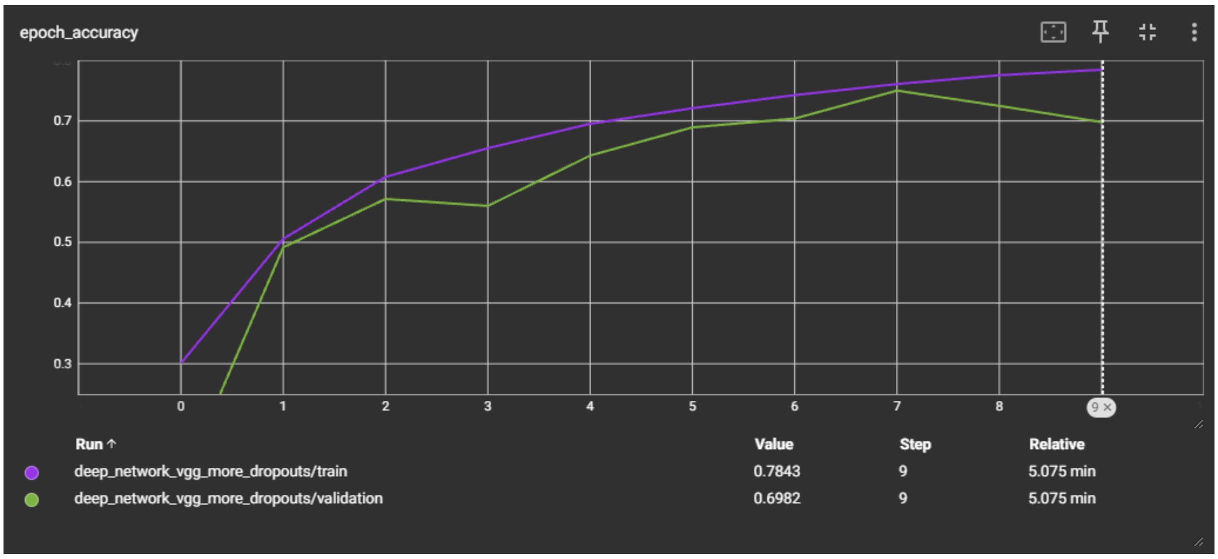
Comment:

The model shows a decent performance during training with an increasing trend in accuracy over epochs. However, the validation accuracy plateaus around 0.79-0.80 after a few epochs, suggesting that the model may be starting to overfit. Additional techniques like data augmentation or regularization may help improve generalization and boost validation accuracy further.

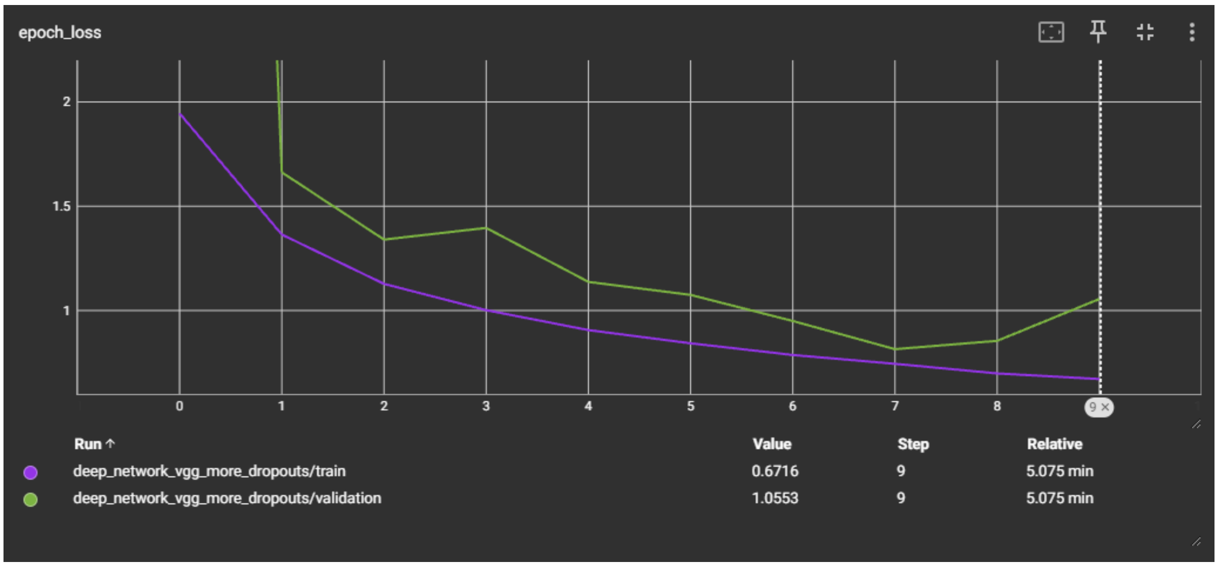
VGG Net: Many Dropouts



```
model = Sequential()

model.add(Conv2D(64, 3, activation='relu', padding='same', input_shape=(32, 32, 3)))
model.add(Conv2D(64, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 2))
model.add(BatchNormalization())
model.add(Dropout(0.5))

model.add(Conv2D(128, 3, activation='relu', padding='same'))
model.add(Conv2D(128, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 2))
model.add(BatchNormalization())
model.add(Dropout(0.5))

model.add(Conv2D(256, 3, activation='relu', padding='same'))
model.add(Conv2D(256, 3, activation='relu', padding='same'))
model.add(Conv2D(256, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 2))
model.add(BatchNormalization())
model.add(Dropout(0.5))

model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 1))  # default stride is 2
model.add(BatchNormalization())
model.add(Dropout(0.5))

model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 1))  # default stride is 2
model.add(BatchNormalization())
model.add(Dropout(0.5))

model.add(Flatten())
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))

model.add(Dense(10, activation='softmax'))
```



	Run ↑	Value	Step	Relative
●	deep_network_vgg_more_dropouts/train	0.7843	9	5.075 min
●	deep_network_vgg_more_dropouts/validation	0.6982	9	5.075 min



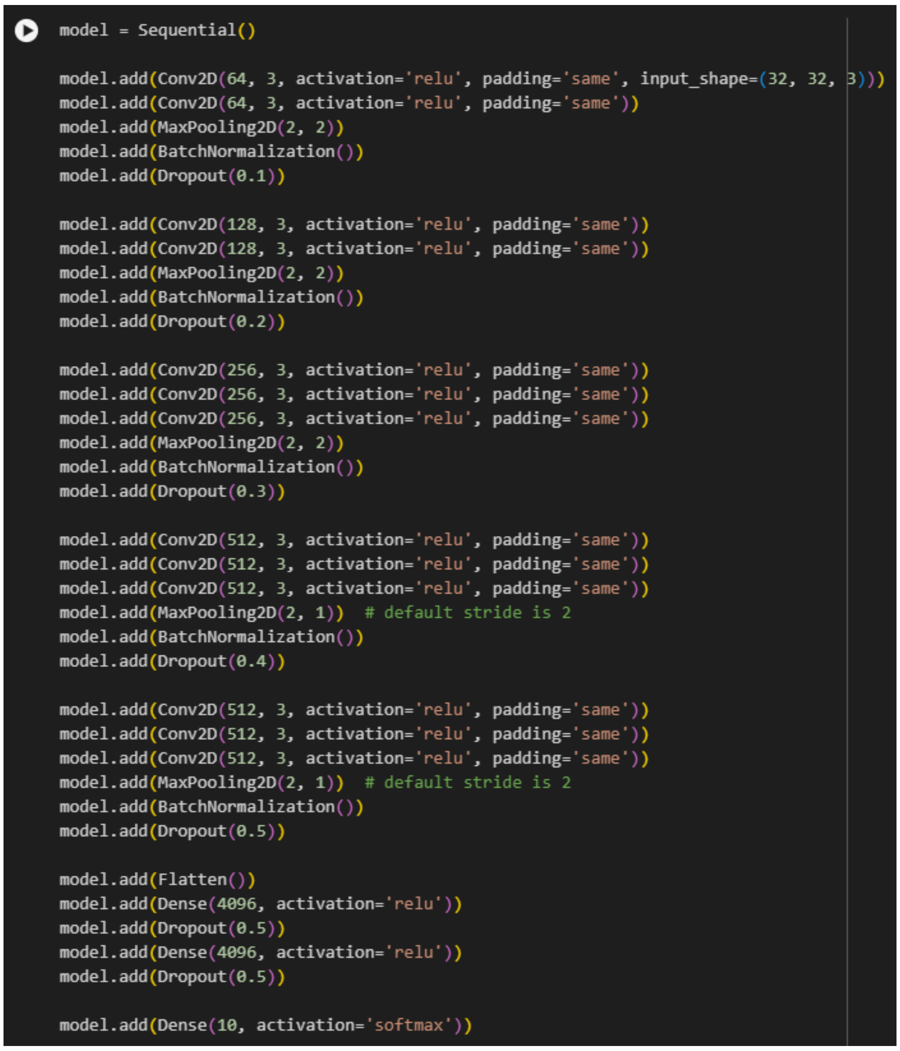
	Run ↑	Value	Step	Relative
●	deep_network_vgg_more_dropouts/train	0.6716	9	5.075 min
●	deep_network_vgg_more_dropouts/validation	1.0553	9	5.075 min

Final Validation Accuracy Score: 0.6982 / 0.7500

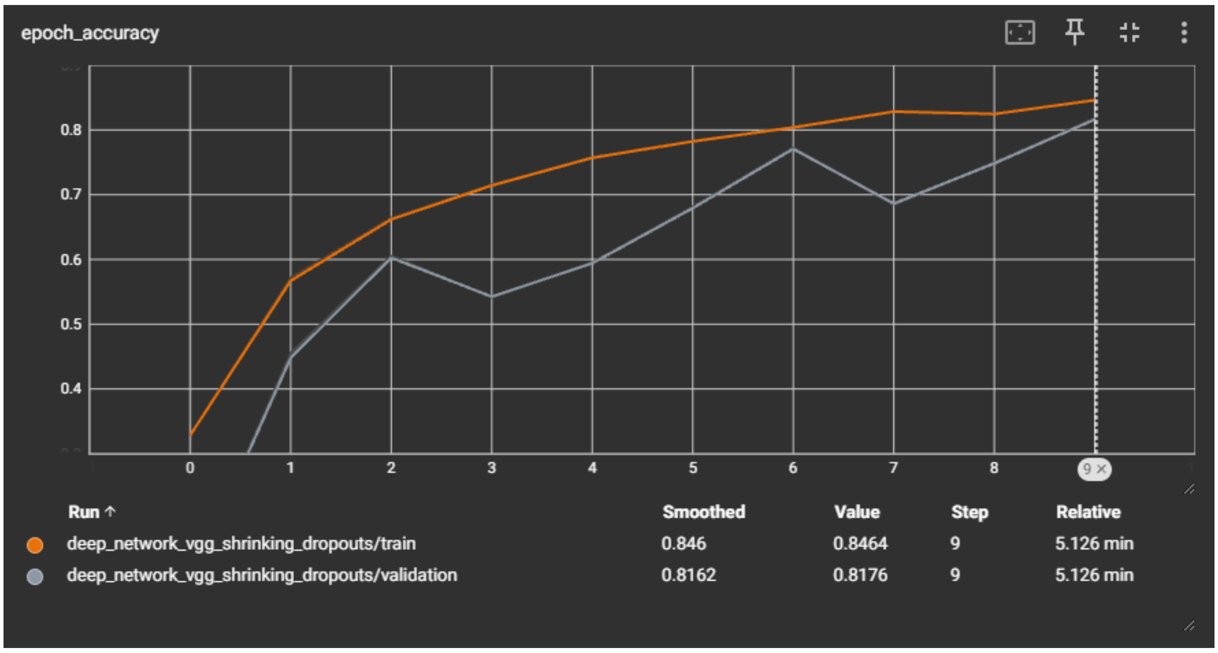
The validation accuracy of the model seems to fluctuate over the epochs, with some epochs showing improvements and others showing slight decreases. Overall, there doesn't seem to be a consistent trend of monotonic increase or decrease in validation accuracy. Instead, the accuracy appears to vary, indicating that the model's performance on the validation set is sensitive to changes in the training process. This variability could be influenced by factors such as the complexity of the model, the choice of hyperparameters, and the characteristics of the dataset. Further analysis and experimentation may be needed to identify the specific factors affecting the validation accuracy and to optimize the model accordingly.

VGG Net: Shrinking Dropouts

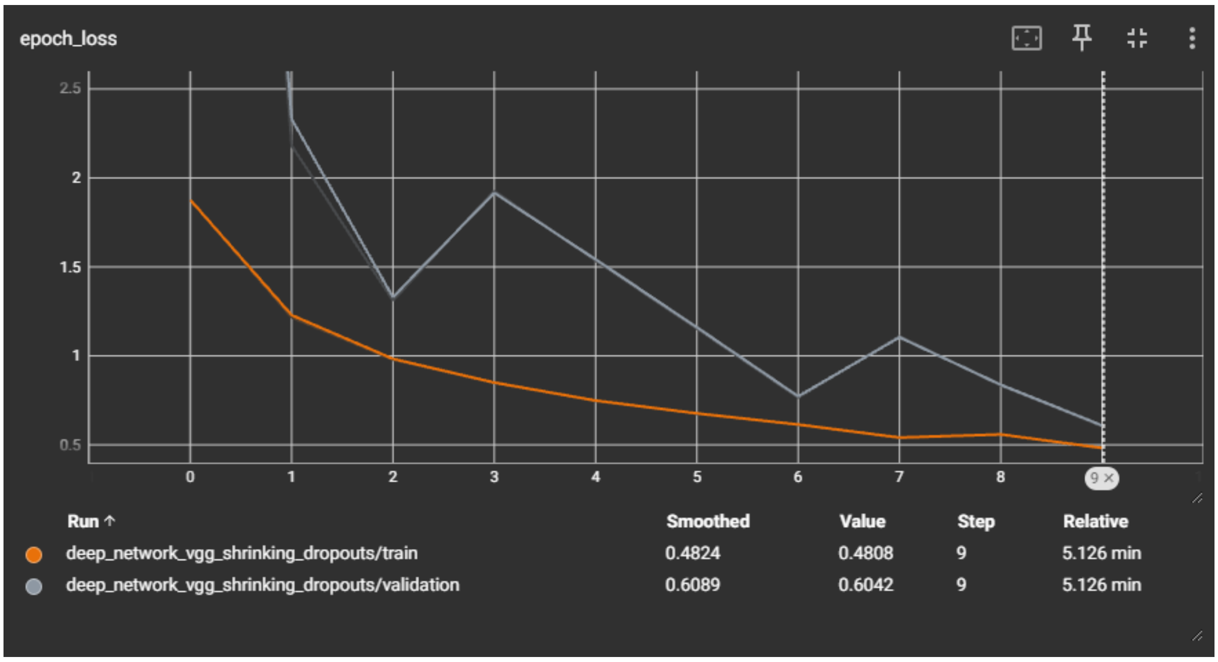
Since previous attempt did not bring expected results I decided to modify dropout strategy by changing how many neuron should we deactivate. This may prevent killing to many of them. At first dropout rate is rather small but eventually it reaches 0.5 – value I choose not to exceed.



```
model = Sequential()

model.add(Conv2D(64, 3, activation='relu', padding='same', input_shape=(32, 32, 3)))
model.add(Conv2D(64, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 2))
model.add(BatchNormalization())
model.add(Dropout(0.1))

model.add(Conv2D(128, 3, activation='relu', padding='same'))
model.add(Conv2D(128, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 2))
model.add(BatchNormalization())
model.add(Dropout(0.2))

model.add(Conv2D(256, 3, activation='relu', padding='same'))
model.add(Conv2D(256, 3, activation='relu', padding='same'))
model.add(Conv2D(256, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 2))
model.add(BatchNormalization())
model.add(Dropout(0.3))

model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 1))  # default stride is 2
model.add(BatchNormalization())
model.add(Dropout(0.4))

model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 1))  # default stride is 2
model.add(BatchNormalization())
model.add(Dropout(0.5))

model.add(Flatten())
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))

model.add(Dense(10, activation='softmax'))
```



Run ↑	Smoothed	Value	Step	Relative
● deep_network_vgg_shrinking_dropouts/train	0.846	0.8464	9	5.126 min
● deep_network_vgg_shrinking_dropouts/validation	0.8162	0.8176	9	5.126 min



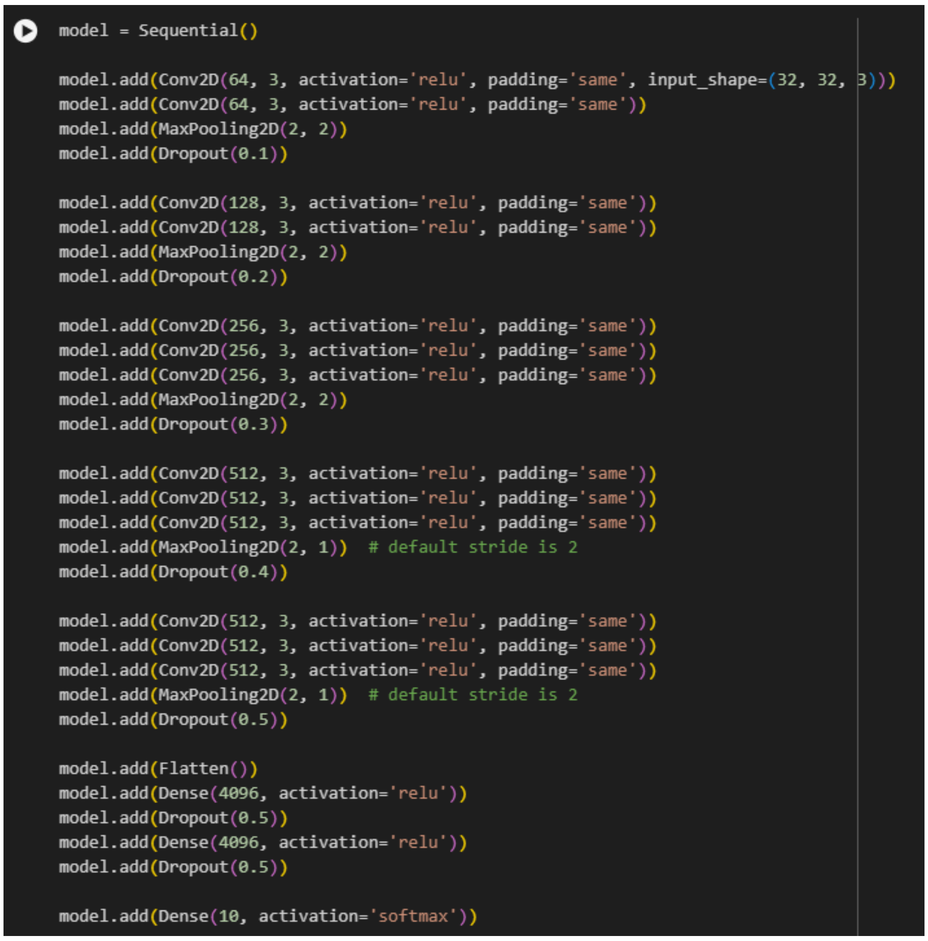
Run ↑	Smoothed	Value	Step	Relative
● deep_network_vgg_shrinking_dropouts/train	0.4824	0.4808	9	5.126 min
● deep_network_vgg_shrinking_dropouts/validation	0.6089	0.6042	9	5.126 min

Final Validation Accuracy Score: 0.8176

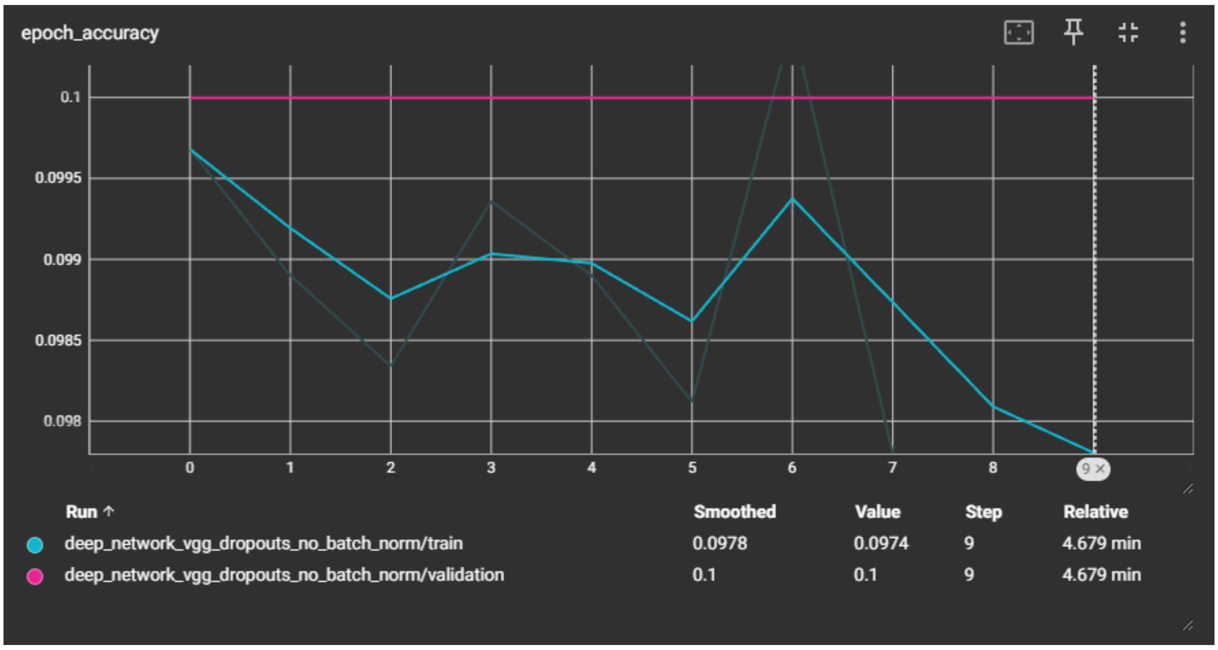
Comment:
As it turned out this intuitive approach brought compelling result by reaching 0.81 with validation accuracy mean while having train accuracy in near proximity starting from 6 epoch. It is my opinion that adding more epochs for this particular model would make it even better.

VGG Net: With dropouts but no Batch Norms

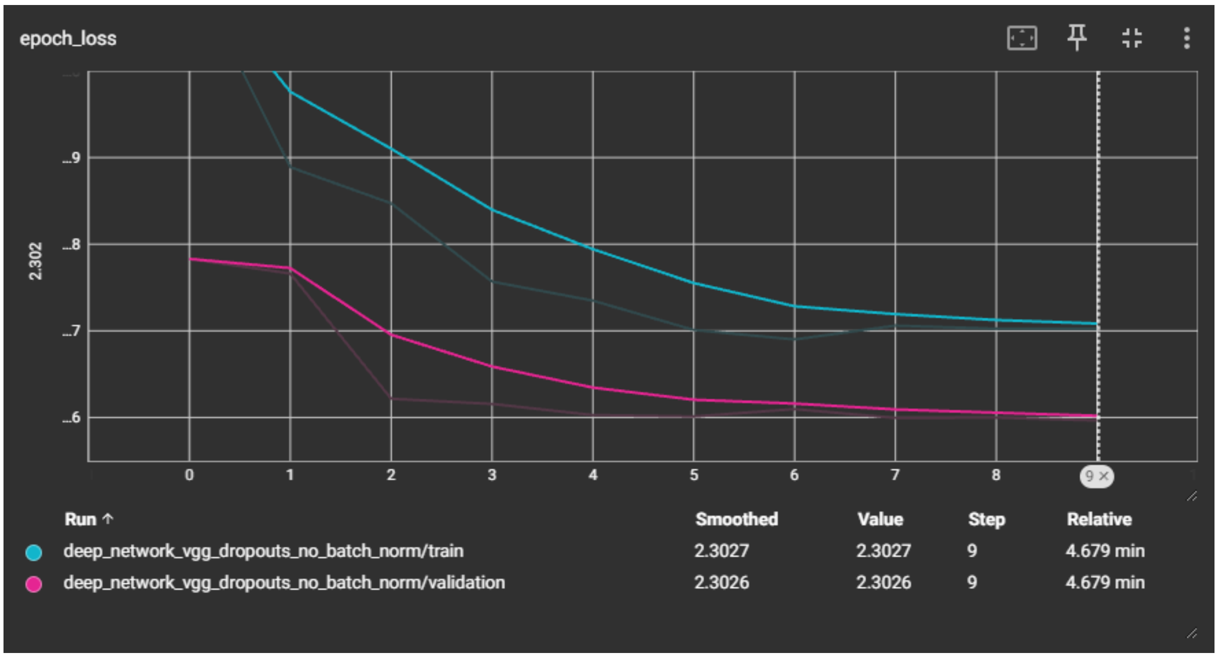
Since last change brought good result, I tried to connect it with idea of not normalizing weights’ vectors since they their imbalance might hold some important information that our model possibly may depend on.



```
model = Sequential()

model.add(Conv2D(64, 3, activation='relu', padding='same', input_shape=(32, 32, 3)))
model.add(Conv2D(64, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 2))
model.add(Dropout(0.1))

model.add(Conv2D(128, 3, activation='relu', padding='same'))
model.add(Conv2D(128, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 2))
model.add(Dropout(0.2))

model.add(Conv2D(256, 3, activation='relu', padding='same'))
model.add(Conv2D(256, 3, activation='relu', padding='same'))
model.add(Conv2D(256, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 2))
model.add(Dropout(0.3))

model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 1))  # default stride is 2
model.add(Dropout(0.4))

model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(Conv2D(512, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, 1))  # default stride is 2
model.add(Dropout(0.5))

model.add(Flatten())
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))

model.add(Dense(10, activation='softmax'))
```



Run ↑	Smoothed	Value	Step	Relative
● deep_network_vgg_dropouts_no_batch_norm/train	0.0978	0.0974	9	4.679 min
● deep_network_vgg_dropouts_no_batch_norm/validation	0.1	0.1	9	4.679 min

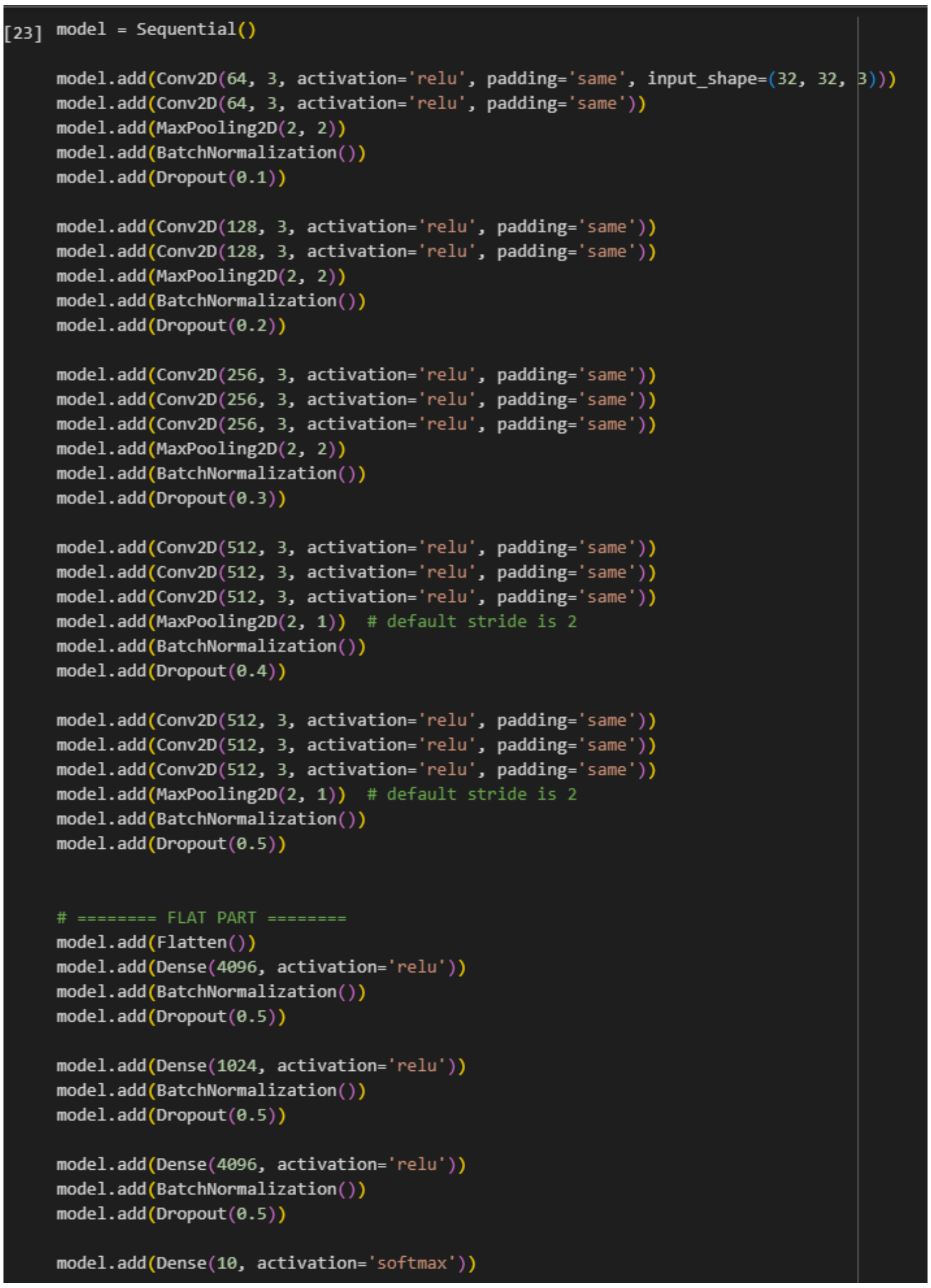


Run ↑	Smoothed	Value	Step	Relative
● deep_network_vgg_dropouts_no_batch_norm/train	2.3027	2.3027	9	4.679 min
● deep_network_vgg_dropouts_no_batch_norm/validation	2.3026	2.3026	9	4.679 min

Final Validation Accuracy Score: 0.1

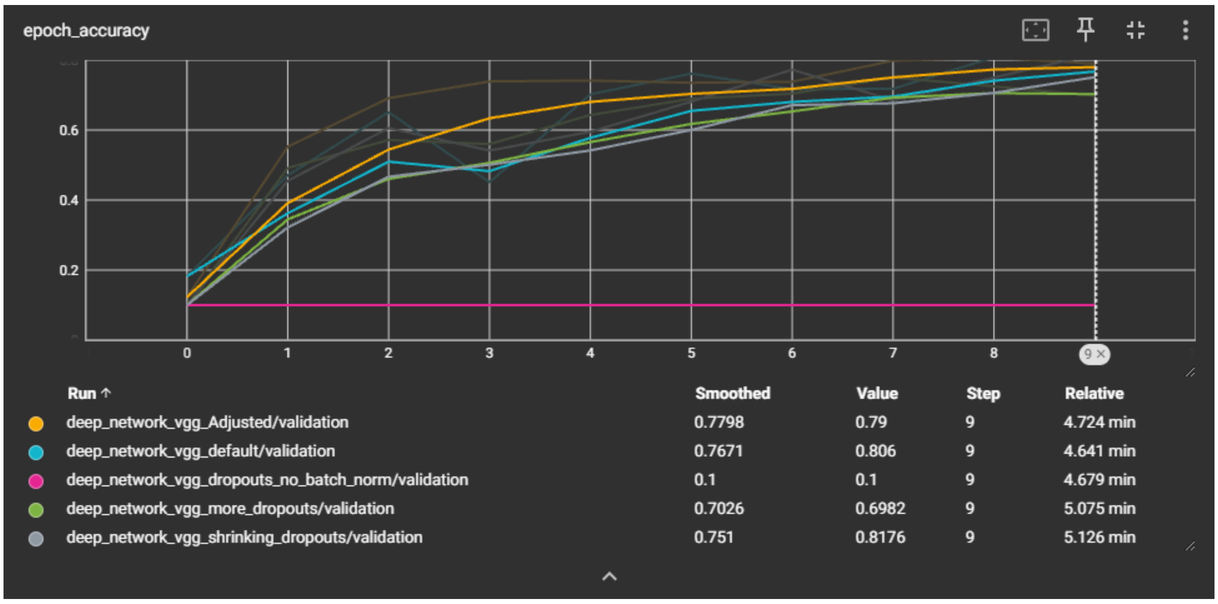
Comment:
The removal of all batch normalization layers led to a significant drop in model performance. The validation accuracy remained stagnant at 10%, indicating that the model was not learning effectively. Batch normalization layers play a crucial role in stabilizing and accelerating the training process by normalizing the activations between layers. Without batch normalization, the model's training dynamics were disrupted, hindering its ability to learn meaningful representations from the data. As a result, the model failed to achieve significant improvement in accuracy, and both training and validation losses remained unchanged throughout the epochs.

VGG Net: Deeper Flattened Part

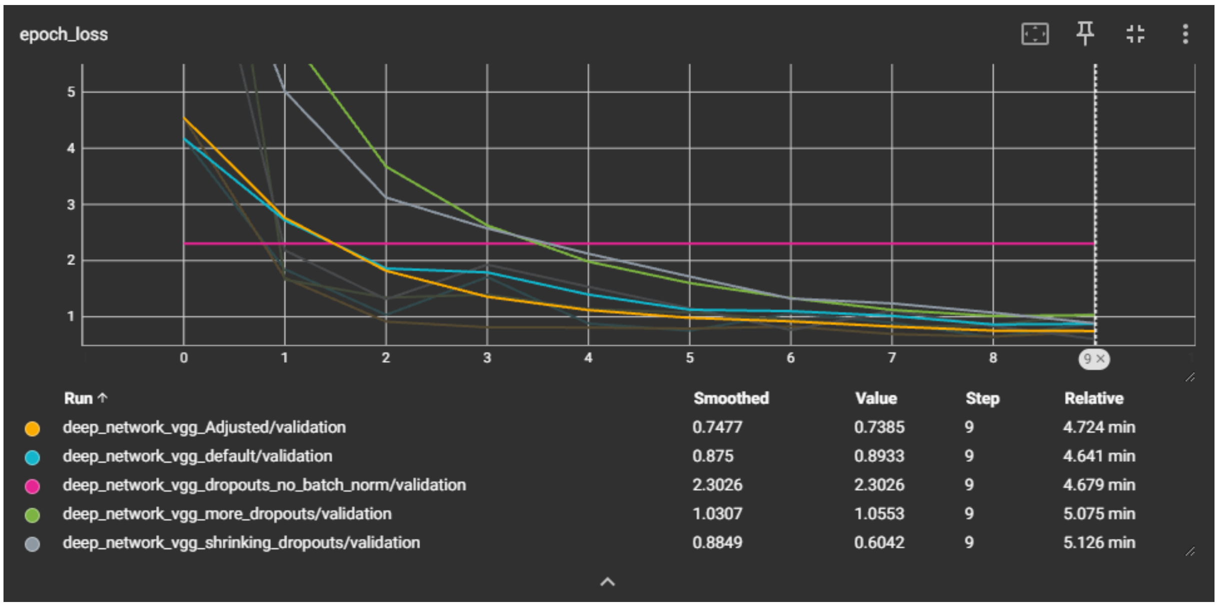


```
[23] model = Sequential()

     model.add(Conv2D(64, 3, activation='relu', padding='same', input_shape=(32, 32, 3)))
     model.add(Conv2D(64, 3, activation='relu', padding='same'))
     model.add(MaxPooling2D(2, 2))
     model.add(BatchNormalization())
     model.add(Dropout(0.1))

     model.add(Conv2D(128, 3, activation='relu', padding='same'))
     model.add(Conv2D(128, 3, activation='relu', padding='same'))
     model.add(MaxPooling2D(2, 2))
     model.add(BatchNormalization())
     model.add(Dropout(0.2))

     model.add(Conv2D(256, 3, activation='relu', padding='same'))
     model.add(Conv2D(256, 3, activation='relu', padding='same'))
     model.add(Conv2D(256, 3, activation='relu', padding='same'))
     model.add(MaxPooling2D(2, 2))
     model.add(BatchNormalization())
     model.add(Dropout(0.3))

     model.add(Conv2D(512, 3, activation='relu', padding='same'))
     model.add(Conv2D(512, 3, activation='relu', padding='same'))
     model.add(Conv2D(512, 3, activation='relu', padding='same'))
     model.add(MaxPooling2D(2, 1))  # default stride is 2
     model.add(BatchNormalization())
     model.add(Dropout(0.4))

     model.add(Conv2D(512, 3, activation='relu', padding='same'))
     model.add(Conv2D(512, 3, activation='relu', padding='same'))
     model.add(Conv2D(512, 3, activation='relu', padding='same'))
     model.add(MaxPooling2D(2, 1))  # default stride is 2
     model.add(BatchNormalization())
     model.add(Dropout(0.5))


     # ======== FLAT PART ========
     model.add(Flatten())
     model.add(Dense(4096, activation='relu'))
     model.add(BatchNormalization())
     model.add(Dropout(0.5))

     model.add(Dense(1024, activation='relu'))
     model.add(BatchNormalization())
     model.add(Dropout(0.5))

     model.add(Dense(4096, activation='relu'))
     model.add(BatchNormalization())
     model.add(Dropout(0.5))

     model.add(Dense(10, activation='softmax'))
```

VGG Net: Results Comparation



	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_vgg_Adjusted/validation	0.7798	0.79	9	4.724 min
●	deep_network_vgg_default/validation	0.7671	0.806	9	4.641 min
●	deep_network_vgg_dropouts_no_batch_norm/validation	0.1	0.1	9	4.679 min
●	deep_network_vgg_more_dropouts/validation	0.7026	0.6982	9	5.075 min
●	deep_network_vgg_shrinking_dropouts/validation	0.751	0.8176	9	5.126 min



	Run ↑	Smoothed	Value	Step	Relative
●	deep_network_vgg_Adjusted/validation	0.7477	0.7385	9	4.724 min
●	deep_network_vgg_default/validation	0.875	0.8933	9	4.641 min
●	deep_network_vgg_dropouts_no_batch_norm/validation	2.3026	2.3026	9	4.679 min
●	deep_network_vgg_more_dropouts/validation	1.0307	1.0553	9	5.075 min
●	deep_network_vgg_shrinking_dropouts/validation	0.8849	0.6042	9	5.126 min

Clearly model with gradually changing dropout layers and batch normalization after each block performed the best of these models. The model with no batch norms proved how crucial that factor actually is.